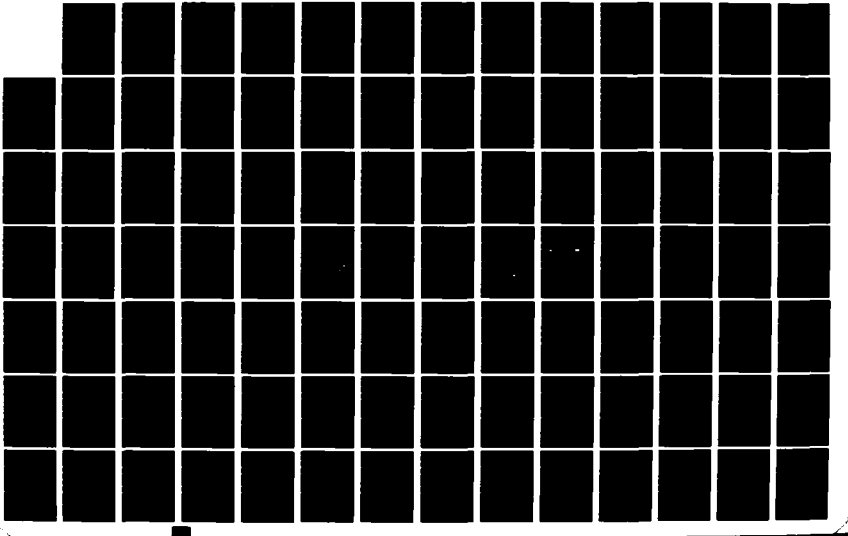
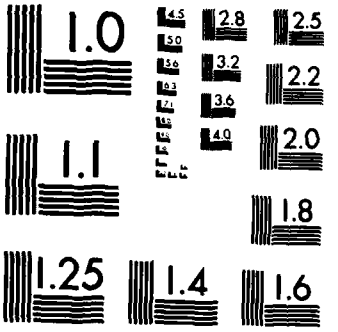
AD-A127 586 LABORATORY FOR COMPUTER SCIENCE PROGRESS REPORT 18 JULY 1/3
1980-JUNE 1981(U) MASSACHUSETTS INST OF TECH CAMBRIDGE
LAB FOR COMPUTER SCIENCE.. M L DERTOUZOS 01 APR 83
UNCLASSIFIED LCS-PR-18 N00014-75-C-0661 F/G 9/2 NL





LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROGRESS REPORT 18

July 1980 - June 1981

Prepared for the

Defense Advanced Research Projects Agency

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: LCS Progress Report 18	2. GOVT ACCESSION NO.: A127586
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE *(and Subtitle)*: Laboratory for Computer Science Progress Report 18 July 1980 - June 1981	5. TYPE OF REPORT & PERIOD COVERED: DARPA/DOD, Progress Report 7/80 - 6/81
	6. PERFORMING ORG. REPORT NUMBER: LCS-PR 18
7. AUTHOR(s): Laboratory for Computer Science Michael L. Dertouzos	8. CONTRACT OR GRANT NUMBER(s): N00014-75-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Laboratory for Computer Science Massachusetts Institute of Technology 545 Tech. Sq. Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE: April 1, 1983
	13. NUMBER OF PAGES: 286
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Office of Naval Research Department of the Navy Information System Program Arlington, VA 22217	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Computation Structures	Information Systems	Office Automation
Computer Networks	Interactive Systems	Personal Computers
Computer Systems	Local Networks	Programming Languages
Dataflow	Message Systems	Real Time Computing
Distributed Systems	Multiprocessing	Specifications
Hardware Protocols	Network Semantics	VLSI
	Network Protocols	Workstations

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The report summarizes the research performed by the MIT Laboratory for Computer Science from July 1, 1980 through June 30, 1981.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

PROGRESS REPORT 18

July 1980 - June 1981

Prepared for the

Defense Advanced Research Projects Agency

Effective date of contract:	**1 January 1979**
Contract expiration date:	**31 December 1980**
Principal Investigator and Director:	**Michael L. Dertouzos** (617) 253-2145

This research is supported by the Defense Advanced Research Projects Agency under Contract No. N00014-75-C-0661

Views and conclusions contained in this report are those of the authors and should not be interpreted as representing the official opinions or policy, either expressed or implied of DARPA, the U.S. Government or any other person or agency connected with them.

This document is approved for public release and sale; distribution is unlimited.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

TABLE OF CONTENTS

ADMINISTRATION

Academic Staff

M. Dertouzos	Director
M. Hammer	Associate Director
A. Meyer	Associate Director

Administrative Staff

J. Badal	Information Specialist
M. Baker	Administrative Assistant
J. Hynes	Administrative Officer
D. Wharen	Assistant Administrative Officer

Support Staff

G. Brown	T. LoDuca
S. Cavallaro	D. Maupin
R. Cinq-Mars	E. Profirio
M. Cummings	P. Vancini

Work reported herein was carried out within the Laboratory for Computer Science, an MIT interdepartmental laboratory. During 1980-81, the principle financal support of the Laboratory has come from the Defense Advanced Research Projects Agency (DARPA), under the Office of Naval Research Contract N0014-75-C-0661. DARPA has been instrumental in the last 18 years and is gratefully acknowledged here.

Assembly and production of this report was done by P. Vancini with assistance from A. Finn.

INTRODUCTION

The Laboratory for Computer Science is an interdepartmental laboratory whose principal goal is research in computer science and engineering.

Founded in 1963 as Project MAC (for Multiple Access Computer and Machine Aided Cognition), the Laboratory developed the Compatible Time-Sharing System (CTSS), one of the first time-shared systems in the world, and Multics -- an improved time-shared system that introduced several new concepts. These two major developments stimulated research activities in the application of on-line computing to such diverse disciplines as engineering, architecture, mathematics, biology, medicine, library science, and management. Since that time, the Laboratory's objectives expanded, leading to research across a broad front of activities that now span four principle areas.

The first such area, entitled Knowledge Based Programs involves making programs more intelligent by capturing, representing, and using knowledge which is specific to the problem domain. Examples are the use of expert medical knowledge for assistance in diagnosis carried out by the Clinical Decision Making Group; the use of mathematical knowledge by the Mathlab Group for an automated "mathematical assistant;" the use of knowledge in programs that comprehend typed natural language (English) queries; and the use of specific knowledge about budget for a budget planning system.

Research in the second and largest area, entitled Machines, Languages and Systems, strives to effect sizable improvements in the ease of utilization and cost effectiveness of computing systems. For example, the Programming Methodology Group strives to achieve this broad goal through research in the semantics of geographically distributed systems. Toward the same goal, the Real Time Systems Group is exploring distributed operating systems and the architecture of single-user powerful computers that are interconnected by communication networks. Communication networks for such distributed environments are pursued by the Computer Systems and Communications Group, while distributed file servers and cryptographic protection techniques are pursued by the Computer Systems Structure Group. Other research in this area includes the architecture of very fast multiprocessor machines by the Computation Structures and Functional Languages and Architecture Groups, and the use of networks to link large numbers of computers engaged in computationally intensive tasks.

The Laboratory's third principal area of research, Theory, involves exploration and development of theoretical foundations in computer science. For example, the Theory of Computation Group strives to understand ultimate limits in space and time

associated with various classes of algorithms, the semantics of programming languages from both analytical and synthetic viewpoints, the logic of programs, and the links between mathematics and the privacy/authentication of computer to computer messages.

The fourth area of Laboratory research, entitled Computers and People, entails societal as well as technical aspects of the interrelationships between people and machines. Examples of research in this area include office automation research carried out by the similarly named Laboratory research group, the use of interconnected computers for planning as well as the sociological impact of computers on individuals, and the ethical problems of distributed responsibility posed by multiprogrammer systems.

During 1980-81, the Laboratory consisted of 279 members -- 39 faculty, 20 visitors and visiting faculty, 74 professional and support staff, 102 graduate and 44 undergraduate students -- organized into 16 research groups. The academic affiliation of most of the faculty and students is with the Department of Electrical Engineering and Computer Science. Other academic units represented in the Laboratory membership are Mathematics, Architecture, Division for Study and Research in Education, Humanities, and the Sloan School of Management. Laboratory research during 1980-81 was funded by 16 governmental and industrial organizations, of which the Defense Advanced Research Projects Agency of the Department of Defense provided about half of the total research funds.

The 1980-81 year was very active. Technical results were disseminated through the publications of the Laboratory members. The following items were the highlights of the year:

The rapid growth and the changing nature of the computer field were felt during the 1980-81 reporting period through the creation of four new research groups. Two of these groups resulted from "mitosis" of the Laboratory's oldest group, Computer Systems Research, which pioneered many of the important early time-sharing results of Project MAC (now the Laboratory for Computer Science). The first of these new groups, entitled Computer Systems and Communications (CSC), is led by Professor Jerome H. Saltzer and includes Principal Research Scientist David D. Clark in its leadership. This group investigates the computer systems problems that arise in making effective use of new communications technology. Current research topics include: gaining field experience with ring networks; the solution of technical problems encountered in building and managing a campus-wide network of several thousand computers; investigation of methods for producing high-quality high-performance implementations of network protocols; and investigation of feasibility of a pocket communications terminal.

The second new research group is entitled Computer Systems Structure (CSS) and is led by Professor David P. Reed. The group's goal is the study and synthesis of system building blocks that support distributed applications. A principal such block currently under construction is a distributed data storage system called SWALLOW. In addition, projects are under way in the areas of: (1) debugging and monitoring distributed systems; (2) intermodule naming and linking in distributed systems; and (3) authentication in distributed systems.

The Educational Computing Group (EC) is the third new Laboratory group established in 1980-81 under the leadership of Professor Harold Abelson with Professors Andrea diSessa and Robert M. Fano as additional senior members. Its goals are to give people personal control over powerful computational resources and to use computation as a catalyst for helping people engage profound ideas from science, from mathematics, and from the art of inteliectual model building. Under DARPA funding, the group's effort is currently focused on the design of an integrated computational environment which can serve as a programming medium for non-expert users. Other ongoing activities center around the role of computation in educational reform, on the impact of computation in changing the nature of the content of science and mathematics curricular at all levels, and on the use of computation in MIT's undergraduate program.

The fourth new group, entitled Functional Languages and Architecture (FLA) is led by Professor Arvind and includes Research Associate Robert E. Thomas in its leadership. Group goals entail the use of dataflow principles as a basis for structuring distributed systems and for exploiting parallel architectures that facilitate the efficient execution of functional language programs including those requiring dynamic resource allocation. Functional languages provide support for advanced programming methods and simplify detection of parallelism in programs. Currently, the group is designing a prototype for a one thousand-processor dataflow machine.

Our major Laboratory focus on geographically distributed systems has continued to occupy the attention of more than half of our staff. We have completed our design of a powerful personal computer that can employ different microprocessors as the technology of the latter progresses. This year, we expect to acquire over 100 such "advanced nodes" which we will use as direct research vehicles in some seven Laboratory research groups, and as general tools for the office automation of LCS.

Our research in distributed computing can be viewed as a search for equilibrium between the opposing forces toward centralization and decentralization -- centralization since it maximizes order by vesting authority in one locus, and decentralization because of people's inherent need to control and use their own resources. We believe that increasing decentralization will have a significant effect on the field of computing in that: (1) it will make possible larger number of inter-

communicating computational resources; and (2) it will permit acceptable operation of the aggregate system in spite of failures of local nodes.

The Laboratory's Distinguished Lecturer Series, initiated in 1976, has proven very successful in attracting members of the MIT community. The 1980-81 lecturers under this series were: Edson D. deCastro (President, Data General Corporation), Richard W. Hamming (Adjunct Professor, Naval Postgraduate School), Grace M. Hopper (Captain, Naval Data Automation Command, U.S. Navy), Richard M. Karp (Professor of Computer Science, University of California at Berkeley), William A. Norris (Chairman and Chief Executive Officer, Control Data Corporation), Louis Pouzin (Director, Pilot Project, Institut National de Recherche d'Informatique et d'Automatique), and William A. Wulf (Professor of Computer Science, Carnegie-Mellon University).

During 1980-81 the Laboratory for Computer Science became the research home of the following faculty members: Assistant Professor Charles E. Leiserson of the Department of Electrical Engineering and Computer Science, Assistant Professors Raviv Kannon and Michael Sipser of the Mathematics Department, and Associate Professor Andrea diSessa of the Division for Study and Research in Education. Mr. John J. Hynes, former Manager of the MIT Department of Nuclear Engineering also joined the Laboratory as Administrative Officer.

New research results developed during 1980-81 were published through the Laboratory's Technical Reports (TR240-TR262) and Technical Memoranda (TM168-TM198), and through several articles in the technical literature.

Michael L. Dertouzos
Director

COMPUTER SYSTEMS AND COMMUNICATIONS

Academic Staff

J.H. Saltzer, Group Leader
D.D. Clark
F.J. Corbató
I. Grief

Research Staff

J.N. Chiappa
E.A. Martin

Graduate Students

R. Baldwin
G. Cooper
S. Curtis
D. Estrin
J. Frankel
S. Kent
L. Lopez
G. Simpson
V. Singh

Undergraduate Students

M. Agbleze
L. Allen
D. Bertko
J. Close
D. Feldmeier
T. Fitzgerald
D. Goldfarb
M. Greenwald
R. Houldin
L. Konopelski
C. Ludwig
S. Meadow
F. Meier zu Sieker
I. Mondori
M. Patton
M. Ross
R. Teal
J. Thomas

Support Staff

D. Fagin
N. Lyall
M. Webber

Computer Systems and Communications

1. INTRODUCTION

At the beginning of 1981, the Computer Systems Research Group split into two parts when David Reed formed the Computer Systems Structure Group. This report mentions some joint work done prior to the first of the year, but in general the work prior to that time has been recorded in whichever group it most naturally fits. As suggested by its name, the Computer Systems and Communications Group whose work is described here is interested in the impact of data communications on computer systems design. Most of the work of this group is experimental in nature, using the MIT local networks, the ARPANET, and the small and large computer systems at MIT as a laboratory. The activities of this group cover a wide range, from hardware design to system programming, and invoke many different levels of network implementation, from physical transmission level to operating system function.

2. RINGNET HARDWARE

A major recent project at the physical transmission level has been the development of a 10 mbit/sec ring network, called the Version 2 ring network (it replaces a 1 mbit/sec, Version 1 ring). The research purpose of this development is to explore the hypothesis that a ring network is equal or superior to the Ethernet in day-to-day operation, maintenance, trouble isolation, and repair properties. At the same time, the development of this network is intended to provide a useful service in interconnecting the Laboratory-developed Nu personal computers and the several DEC PDP-11, LSI-11 and VAX-11 computers of the Laboratory.

The Version 2 ring hardware is being marketed by Proteon Associates of Waltham, MA. The ring interface consists of two parts, a net control/modem card and a host-specific interface to the PDP-11 UNIBUS. This set of two cards, at a price of $3200, is now in production, and has been shipped to several different customers. Other host-specific interface boards are under development in our Laboratory for the NuBus and the S-100 bus, under the supervision of Cliff Ludwig, while Proteon Associates is developing one for the DEC Q-bus and exploring one for the Intel Multibus.

As part of this project, Glenn Simpson developed an alternative modem design that is completely digital, as opposed to the analog phase lock loop design that was produced for the modem by Proteon. The digital design has the advantages that it is

more obviously reducible to VLSI, and it sidesteps some stability issues which had to be addressed in the closed loop analog modem.

As part of the ring interface development a large battery of test programs were developed by Liza Martin and Larry Allen. These programs have now been released to other sites that are procuring the ring hardware.

A group of graduate students, as a class project, designed a VLSI chip containing a simplified, low-performance version of the ring network control circuitry. The chip will be fabricated and tested this summer, though it should be realized that this design was done as a feasibility study rather than with serious intent of production use.

3. MIT INTERNET

Internetwork interconnection is a second major area of activity of this group, since there are now four high-speed local networks in the building at 545 Technology Square, as well as the ARPANET. Again in this area multiple purposes are being served. The research goal is to understand how to extend present interconnection techniques to a scale where perhaps 100 local networks and 5000 computers can communicate at high speeds. The anticipated data communication requirements of the MIT campus in 1990 serve as the focus for this interest. At the same time, service-producing results of the work are of immediate interest to other Laboratory users.

A permanent high-speed connection was established between the LCS ring net and the ARPANET, using a new C/30 IMP that was delivered for this purpose. There are now two net interconnection gateways in operation, one between the ARPANET and the 1 mbit/sec ring net, and the other connecting the 1 mbit/sec ring, the Ethernet and the Chaosnet. Both are implemented using Digital LSI-11 computers. Substantial effort has been invested in stabilizing the service provided by these two machines, including an auto-restart facility, developed by Larry Allen, which reboots the machines automatically over the network in case of software crash. The frequent outages which were initially observed with these interconnect machines now appear to be largely eliminated.

New software for these machines, which incorporates more sophisticated routing algorithms and better maintainability and modifiability, has been coded in C, and is being prepared for installation now by Noel Chiappa.

We have worked with a committee being chaired by Prof. F.J. Corbató to plan for the networking of MIT. The primary output of this committee has been a

memorandum outlining the scope of the data communication problem at MIT and specifying the general requirements a network must meet.

4. NEW PROTOCOL DEVELOPMENT

David Clark continues to participate in the ARPA working group developing Internet and Transmission Control Protocol (TCP). These protocols have now been adopted as DOD standards for internetting, and are beginning to be placed in service here at MIT as well as elsewhere in the ARPA community. A number of projects are underway to explore new protocols. In particular, for page or record level file access across a network, we have developed a protocol named Simple File Access Protocol (SFAP), which might be suitable for support of remote files for personal computers. Geoffrey Cooper, as a master's thesis, is exploring possible implementation techniques for SFAP. In particular, he is investigating the question of whether layering SFAP into a reliable datagram protocol and file transfer superstructure is of any benefit, or whether the layering is in fact an artificial structure which produces inefficiencies and bulkiness in the code.

A related area of protocol, the control of routing by specifying the complete route when a packet enters the network, so as to simplify and speed up forwarding nodes, is the subject of a master's thesis in progress by Vineet Singh. In this thesis, Singh is developing algorithms by which administratively distinct regions of a network can maintain local routing information and cooperate with other regional routing services to materialize a complete route for any given connection.

5. PROTOCOL PERFORMANCE

One of our principal research goals for the year is to determine how TCP will perform given the wide range of networks over which it is intended to operate. In particular, we are interested in how the performance characteristics of the protocol impact on the implementation done inside the host machine. In order to explore this, we have extensively metered the implementation of TCP on Multics, and we have produced a number of implementations of TCP for the Xerox Alto desktop computer, which implementations have been used for performance experiments as well as for service. The conclusions of our preliminary study are presented in a document now in preparation, but the general results are that: 1) the details of TCP are not a material contributor to overhead observed. The actual protocol processing is a very small part of the observed cost. The most important single host-related cost of TCP is that the packet is checksummed in software, a somewhat expensive operation whose benefit has been clearly demonstrated. 2) We have shown that the classical flow control mechanism, windows, does not necessarily work well under all

circumstances in which TCP is expected to operate. When the total delay in the net is substantially more than the buffer space available in the receiving host, it is difficult to make windows work in such a way that high transmission rate is achieved. Mismanagement of windows, which we have identified by the term Silly Window Syndrome, is a major contributor to the abysmal throughput sometimes observed in the internet. 3) We have demonstrated that operating system overhead is a principal contributor to poor performance of host protocol implementations. Especially in classical operating systems, which were not intended to be used as communications processors, process switching is sufficiently costly that dealing with individual incoming packets causes a very large cost in process management. David Clark, Liza Martin, Larry Allen, and Geoffrey Cooper are currently exploring what might be appropriate structures for operating systems that should operate well in a network environment.

6. MULTI-PROTOCOL COMPUTER MAIL

The ARPANET mail protocols do not permit delivery of mail to computers connected to nets other than the ARPANET. The ARPA community has proposed a new protocol, called Mail Transfer Protocol (MTP) which operates on top of TCP to provide an internet mail service. A protocol converting forwarded mail, developed by David Clark, is now in operation on the Multics system, which makes it possible to send and receive mail to systems such as the RTS 11/70 which is connected only to the local net. Larry Allen has also produced a mail package for the UNIX system, which is capable of using the Multics mail forwarder. This makes it possible to transfer mail between UNIX and systems on the ARPANET not yet upgraded to the new mail protocol.

7. HAND-HELD TERMINAL

This year. David Clark started a project to produce a portable terminal small enough to fit in a pocket. The goal is to exploit the technology and packaging that is now available and being used in various sorts of pocket computers and large calculators. The fundamental idea of this research is that a pocket terminal will only be really usable if the application program is specifically programmed to deal with this class of terminal. Our first test application has been in receiving and sending mail. We have programmed an Alto display to simulate the eventual terminal display, a single line of 36 LCD characters, and have put together various packages to see how one might read text through such a display. Cliff Ludwig is also developing a first version of a hardware prototype, which is not packaged as a small unit, but which has the correct sort of display, so that we can experiment with the actual technology.

8. OTHER ACTIVITIES

A new project was initiated by Jerome Saltzer to explore the possibility of a two-way community cable TV distribution system as a high-bandwidth data communication path to private homes. Discussions have been held with Continental Cablevision of Massachusetts concerning the possibilities of setting up an experimental data service on the CATV system that is about to be installed in the city of Newton. Deborah Estrin has completed an initial study of the technical feasibility of using Ethernet-like CSMA/CD for control of access in such a system, with the general conclusion that CSMA/CD will work well unless most traffic is small packets such as terminal input. (This same conclusion applies to the 10Mb/sec Xerox Ethernet.) Discussions are underway with potential modem manufacturers with the goal of initiating a joint project with one of them.

A distributed PASCAL compiler is being implemented by James Frankel as part of a Harvard University Ph.D. dissertation to explore the problems of distributing a large, predictable computation over several network-connected computers. A completely dynamic resource allocation scheme is being used, in which the compiler explores the network to find out how many free computers are around, then it assigns parts of the compilation computation task accordingly.

Publications

1. Saltzer, J., Reed, D. and Clark, D. "Source Routing for Campus-Wide Internet Transport," IFIP Working Group 6.4 Workshop, Workshop on Local Area Networks, Zurich, Switzerland, August 1980.

2. Saltzer, J., Reed, D. and Clark, D. "End-to-End Arguments in Systems Design," Proceedings of Second International Conference on Distributed Computing Systems, Paris, France, April 1981, 509-512.

3. Kent, S. "Protecting Externally Supplied Software in Small Computers," MIT/LCS/TR-255, MIT Laboratory for Computer Science, Cambridge, MA. September 1980.

Theses Completed

1. Fitzgerald, T. "The Implementation of a File Transfer Protocol on Multics," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

2. Kent, S. "Protected Externally Supplied Software in Small Computers," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1980.

3. Thomas, J. "A Multi-Protocol Network Mail Transport Facility for Multics," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1980.

Theses in Progress

1. Baldwin, R. "An Evaluation of the Recursive Machine Architecture," S.M. and S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1982.

2. Houldin, R. "Formats and Controls for a One-Line Computer Terminal Display," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1981.

3. Lopez, L. "Gateway Congestion Control," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1982.

4. Ludwig, C. "An Implementation of a Portable Personal Terminal," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1982.

5. Martinez, D. "A Central Switcher for Line Switched Message Oriented Computer Input/Output," S.M. and S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1981.

6. Meier zu Sieker, F. "A Telex Gateway for the Internet," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1981.

7. Simpson, G. "NEMO- A Monitoring Station for a Local Area Network," S.M. and S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected February 1982.

8. Singh, V. "A Routing Service for Campus-Wide Internet Transport," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

Talks

1. Clark, D. "On the Limits of Distributed Atomic Update,"

   Workshop on Fundamental Issues in Distributed Computing, Pala Mesa, CA, December 1980;
   5th Berkeley Workshop, Berkeley, CA, February 1981.

2. Saltzer, J. "Why A Ring?" Xerox Palo Alto Research Center, Palo Alto, CA, December 1980.

3. Saltzer, J. "End-to-End Arguments in System Design,"

   University of Cambridge, Oxford, England, August 1981;
   Workshop on Fundamental Issues in Distributed Computing, Pala Mesa, CA, December 1980;
   IBM San Jose Research Laboratory, San Jose, CA, December 1980;
   INRIA Workshop on Distributed Computing, Rocquencourt, France, April 1981.

4. Saltzer, J. "Rings, Ethernets and Broadband, the Underpinnings of Local Networks," New York Academy of Sciences, NY, May 1980.

5. Saltzer, J. "Evolution of Distributed Computer Systems," Computer Science Conference Address, National Security Agency, Washington, DC, May, 1980.

Committee Memberships

Clark, D., DARPA IPTO Internet TCP Working Group

Chiappa, N., DARPA IPTO Internet TCP Working Group

Martin, E., DARPA IPTO Internet TCP Working Group

Saltzer, J., DoD/DDRE Security Working Group Member

COMPUTER SYSTEMS STRUCTURE

Academic Staff

D.P. Reed, Group Leader

L. Svobodova

Graduate Students

G. Arens
W. Gramlich
W. Hamscher
A. Mendelsohn
R. Schiffenbauer
K. Sollins
D. Theriault
C. Topolcic

Undergraduate Students

D. Daniels
C. Leckband
J. Marggraff
P. Ng
M. Novick
J. Stamos
K. Yelick

Support Staff

D. Fagin

Computer Systems Structure

1. INTRODUCTION

The Computer Systems Structure Group was created in January 1981 from part of the Computer Systems Research Group. The work reported here was partially done in that context.

During the past year, we have had two major foci of activity--distributed data storage systems (the SWALLOW project) and debugging methods for distributed applications. Our work in these two areas is described in the following two sections.

In addition, we began study in 1981 on the problem of using encryption-based methods to protect data in a distributed system. This study is gradually acquiring some momentum and manpower, and we expect it to be a more significant part of next year's progress report. Here we report on early efforts.

2. THE SWALLOW DISTRIBUTED DATA STORAGE SYSTEM

During the past year, we have concentrated on constructing the SWALLOW repository prototype. The repository is a shared data storage server. Some features of the repository design are:

- integrated use of optical (write-once) storage
- efficient storage of small objects
- support for atomic actions
- high reliability
- low access time.

During the past year, we feel we have made significant contributions particularly in the management of storage for small objects in write-once, highly stable secondary storage. David Reed, Liba Svobodova, and Gail Arens have created a new storage organization called append-only version storage that speeds access, minimizing work space, and speeds recovery from crashes. Other work has included study of small object management for the SWALLOW broker, study of cryptographic security for the broker/repository, interaction, and development and testing of SMP, a highly-optimized, problem-oriented protocol based on the ARPA User Datagram Protocol.

3. DEBUGGING IN A DISTRIBUTED SYSTEM

This project is intended to produce tools and concepts for debugging and testing distributed application.

During 1980, Robert Schiffenbauer developed a tool that is intended to help in debugging low-level protocols between machines. His S.M. thesis is now completed. The basic idea of this work was to provide a debugging/monitoring station that mediates all communication between nodes of an application--displaying messages sent, controlling order and time of delivery, etc. To be helpful, the debugging station can slow down or stop the execution of nodes, to bridge the gap between thinking speeds and computer speeds.

Wayne Gramlich has begun working on generalizations of these tools to handle higher-level interactions (involving groups or patterns of messages) and controlling/monitoring activities inside nodes.

4. SYSTEMS ASPECTS OF ENCRYPTION-BASED PROTECTION

During the past year, Dean Daniels has built a prototype authentication server that runs as a process on Tops-20. We intend to augment a number of protocols, such as TFTP and SMP, so that they optionally will use the authentication server to initiate secure communications.

Publications

1. Arens, G. "Recovery of the SWALLOW Repository," MIT/LCS/TR-252, MIT Laboratory for Computer Science, Cambridge, MA, January 1981.

2. Liskov, B., Herlihy, M.H., Leitner, G. and Sollins, K.R. (Ed.), Report on the Workshop on the Fundamentals of Distributed Computing, Fallbrook, CA, December 1980, to appear Operating Systems Review and SIGPLAN Notices, 1981.

3. Reed, D.P. "Atomicity in Distributed Systems," Report on the Workshop on the Fundamentals of Distributed Computing," Fallbrook, CA, December 1980, to appear in Operating Systems Review and SIGPLAN Notices, 1981.

4. Reed, D.P. "Implementing Atomic Actions on Decentralized Data," ACM Seventh Symposium on Operating Systems Principles, Pacific Grove, CA, December 1979, to appear in CACM, 1981.

5. Reed, D.P. and Svobodova, L. "SWALLOW: A Distributed Data Storage System for a Local Network," to appear in Proceedings of IFIP W.G. 6.4 Workshop on Local Area Networks, Zurich, Switzerland, August 1980.

6. Saltzer J.H., Reed, D.P. and Clark, D.D. "Source Routing for Campus-Wide Internet Transport," to appear in Proceedings of IFIP W.G. 6.4 Workshop on Local Area Networks, Zurich, Switzerland, August 1980.

7. Saltzer, J.H., Reed, D.P. and Clark, D.D. "End-to-End Arguments in System Design," Proceedings of the Second International Conference on Distributed Computing, Paris, France, April 1981, 509-512.

8. Sollins, K.R. "Copying Structured Objects in a Distributed System," Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, CA, February 1981, 284-300, and to appear in Computer Networks, 1981.

9. Svobodova, L. "Management of Object Histories in the SWALLOW Repository," MIT/LCS/TR-243, MIT Laboratory for Computer Science, Cambridge, MA, July 1980.

Theses Completed

1. Arens, G. "Recovery of the SWALLOW Repository," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1981.

2. Chiang, A. "Performance of Viewdata System," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

3. Dagresta, J. "A People Locating System," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

4. Maloney, J. "A Distributed File Server Which Supports Replicated Files," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

Theses in Progress

1. Greenwald, M. "A Graphics Interface for CLU," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected July 1981.

2. Ketelboeter, V. "Forward Recovery in Distributed Systems," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1981.

3. Lederman, A. "A PASCAL Structured Oriented Editor," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1981.

4. Mendelsohn, A. "A Framework for User Interfaces to Distributed Systems," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1981.

5. Schiffenbauer, R. "Debugging in a Distributed System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1981.

6. Stamos, J. "Grouping Strategies for an Object Oriented Virtual Memory," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected February 1982.

Conference Participation

1. Reed, D.P. "Atomicity in Distributed Systems," Workshop on the Fundamentals of Distributed Computing, Fallbrook, CA, December 1980.

2. Reed, D.P. "Protection and Abstraction in Distributed Systems," Workshop on Complete Distributed Systems, INRIA, Rocquencourt, France, April 1981.

3. Reed, D.P. "The SWALLOW Distributed Data Storage System," Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, CA, February 1981.

4. Reed, D.P. and Svobodova, L. "SWALLOW: A Distributed Data Storage System for a Local Network," IFIP W.G. 6.4 Workshop on Local Area Networks, Zurich, Switzerland, August 1980.

5. Saltzer, J.H., Reed, D.P. and Clark, D.D. "End-to-End Arguments in System Design," Second International Conference on Distributed Computing Systems, Paris, France, April 1981.

6. Saltzer, J.H., Reed, D.P. and Clark, D.D. "Source Routing for Campus-Wide Internet Transport," IFIP W.G. 6.4. Workshop on Local Area Networks, Zurich, Switzerland, August 1980.

7. Sollins, K.R. "Copying Structured Objects in a Distributed System," Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, CA, February 1981.

8. Sollins, K.R. "Name Server," Workshop on the Fundamentals of Distributed Computing, Fallbrook, CA, December 1980.

Talks

1. Reed, D.P. "Local Area Networks--Ether and Ring Nets," ComputerVision, Bedford, MA, February 1981.

2. Reed, D.P. "Organization of the SWALLOW repository," Xerox Palo Alto Research Center, Palo Alto, CA, February 1981.

3. Reed, D.P "SWALLOW: A Distributed Data Storage System for a Local Network,"

INRIA, Rocquencourt, France, August 1980;
GTE Telenet Communication Corporation, Vienna, VA, November 1980.

EDUCATIONAL COMPUTING

Academic Staff

H. Abelson, Group Leader
A. diSessa
R.M. Fano

Research Staff

D. Neves

Graduate Students

S. Bagley

Undergraduate Students

J. Dempsey
G. Kiczales

Support Staff

B. J. Savage

Educational Computing

1. INTRODUCTION

The Educational Computing Group was formed in January 1981. The goals of the group are to develop effective ways to give people personal control over powerful computational resources and to use computation as a catalyst for helping people engage profound ideas from science and from the art of intellectual model building.

During the past five months, we have begun work on two substantial projects. The first is an experiment on the use of computation in teaching mathematics and physics to MIT undergraduates. We believe that computation can provide a perspective that makes it possible to express ideas from science and mathematics in terms of constructive, process-oriented formulations that are more assimilable, more in tune with intuitive modes of thought, than the axiomatic-deductive formalisms in which these ideas are usually couched. (Our recently published book presents substantial computational reformulations of topics in mathematics.[1]). In the coming semester, we will conduct a preliminary teaching experiment during which we will attempt to document how suitably designed computation-rich environments can engender a different quality of learning experienced by undergraduates.

Our second, and major focus of activity these past few months has been to begin the design and implementation (under DARPA funding) of an integrated computational tool for non-expert users. This is meant to be a powerful general-purpose programming environment that includes capabilities for text-editing, programming, data manipulation and inter-user communication. The remainder of this report focuses on this aspect of our work, although we point out that this effort and the preceding one are closely tied. For one thing, the existence of a powerful, yet easy to use computational environment will facilitate more significant efforts in educational computing than have heretofore been possible. At the same time, our projections for educational use supply clear images that guide us in developing the general system, and also serve as testing grounds for preliminary implementations.

Here is a summary of our progress to date on the integrated system:

- We have outlined on the semantics for the underlying language and implemented (most of) an interpreter for it on the LISP machine.

- We have begun the design of a graphics-based user interface, which we hope to complete by the end of the summer.

- We have implemented (within LISP, on the LISP machine) working examples of the kinds of applications we expect the integrated system to support.

These items are discussed in detail in the remainder of this report, whose outline is as follows: We begin by setting forth some general principles which have guided us in the design of a system for non-expert users. We then sketch the system, called "Boxer", as we currently envision it. The next section of the report briefly describes the application examples. Finally, we outline the next steps in the implementation effort as we see it and our projected activity over the coming months.

2. DESIGNING SYSTEMS FOR NON-EXPERT USERS

Before presenting the Boxer system, we comment upon some of the principles that continue to shape its design, principles we regard as important to keep in mind when building computer-based tools for non-expert users. To be sure, when these principles are expressed in their general form, they can easily be regarded as "motherhood issues." On the other hand, our Boxer system is rather different from other programming environments, either proposed or in existence, and this difference arises (in our view) from adhering to these principles with a greater than usual degree of consistency. In the paragraphs below we will try to point out instances where we feel that general principles have dictated design choices that are out of the ordinary, or even controversial.

2.1. Integration and Coherence

A major criterion for our system is that it be integrated and coherent, that is, it should appear to the user as a single entity rather than as a collection of special-purpose subsystems. Our proposal to DARPA [2] dealt with the importance of integration in designing a system for non-expert programmers, and we shall not repeat the arguments here. Instead, we focus on some of the design implications of integration.

From the point of view of coherence it is extremely undesirable (even inexcusable) if the editor used to compose and modify programs has very different functionality than the editor used to compose and modify text, or the editor used to compose and modify mail and messages, or the editor used to alter file directories, and so on. A better approach is to a view the system as not having an "editor" at all, but rather to incorporate capabilities for manipulating text into the standard user interface. Or, as another way to say the same thing, one should regard all system commands as being issued from the editor, which is equipped with an "editing" command called "do-it"

that causes an indicated piece of text to be executed. (Such a system has been incorporated into the LISP machine ZTOP editor interface by Henry Lieberman.) We plan to build on this work implementing Boxer. This "editor-top-level" notion also obviates the need for a separate system monitor, and encourages one to regularly incorporate all "monitor" functions into the user's programming language. Another implication of the ability to type a piece of text and then "do-it" is that the user language should be interpreted (or else have a *very* smart incremental compiler).

Consonant with the editor-top-level notion is the idea that there should be a uniform convention for dealing with any text that appears on the screen. In Boxer, any text appearing on the screen, whether typed by the system, typed by the user, previously executed or not, is available to be manipulated, edited, or re-executed with "do-it." This principle of "what you see is what you have" enables a mode of program construction, known as "concrete programming," whereby the user types and executes statements one by one and then at some later time indicates that the typed statements (possibly after editing them) should be incorporated into a program. This on-the-fly programming methodology has the further integrative effect of minimizing the distinction between constructing a program and running it. (One previous system which makes extensive use of concrete programming is David Smith's "Pygmalion" [3], developed at Xerox PARC. Much closer to Boxer is the "Tinker" system [4]currently being developed at MIT by Henry Lieberman.). It is interesting to note that the concrete programming approach is at odds with the fashionable idea that programming environments for non-expert users should contain a healthy dose of "top-down structured programming" medicine to encourage careful design and specification steps as precursors to implementation.

Another opportunity to adhere to uniform conventions is in the naming of objects. Simple data, compound data, programs, and files (or rather, environments,for we will see in the paragraph below that Boxer has no files per se) should all be named and referenced according to the same mechanism. Thus Boxer, like LISP, includes compound data and procedures as "first-class data objects." However, unlike MACLISP, it does not maintain separate name spaces for procedures and variables. In addition, this uniform naming discipline, together with the incremental style of concrete programming, goes in our view entirely against the popular notion that novice users should be subjected languages with enforced strong data typing of variables. Indeed, we would like the user to regard naming as simply assigning names to *things*, whose attributes can be readily modified with a minimum of programming overhead. Moreover, we would prefer to keep opaque to the user such traditional "computer science" concerns as the difference between a list, an array, and a table, between a procedure and the text that represents it or the file in which it is stored, not to mention the distinction between a number and a character string that happens to consist entirely of digits.

A final area for integration is the abolition of any separate notion of "file system." From the user's point of view, a file system serves two main purposes. The first is to provide some or large-scale organization for the various projects the user may be working on. The second is to provide some degree of session-to-session permanence for interactions and program constructs. In Boxer, we will see that the first function is incorporated into the general hierarchical structure provided by the system (and the "addressing" function normally played by a file directory is taken over by the regular system naming mechanism). The second function is provided by the capability for the user to specify portions of the environment as "permanent" or "temporary." Our hunch is that even this is unnecessary for most naive users, who will use Boxer with the default that everything is "permanent," i.e., will not go away between sessions unless explicitly deleted. We recognize, of course, that there is room for much more sophisticated environment management systems such as Goldstein and Bobrow's PIE [5]. But we do not consider this to be an important concern for novice users.

2.2. Compatibility with Simple Mental Models

Integration can be viewed as the notion that a system's user should be able to understand its performance in terms of a small number of mental models. Beyond that, one should demand that these models be relatively simple. One of the most important criteria for a system that is meant to be a versatile tool for non-expert users is that it should be easy for the user to feel *in control* of it.

In considering "simple mental models" of system performance, we have found it useful to employ the distinction between "surrogate models" and "task action models" that was brought to our attention by Richard Young in his investigation of the mental models employed by users of hand-held calculators [6]. Young's notion of a surrogate model is as a coherent mental picture from which one can deduce most behavior of the machine. For example, when we describe the procedure call mechanism in terms of substitution, or contours, or the workings of an interpreter, we are employing a surrogate model. In order to enable users to reason about and to predict system behavior, it is of course desirable to design the system so that it can be understood in terms of simple surrogate models. But one should realize that such simplicity is often a mathematician's simplicity, the statement that a small set of axioms suffices, through possibly long logical chains, to explain system behavior. Most novice programmers are not mathematicians, and the impact of a simple surrogate model may be less than one would suppose. In dealing with non-expert users it is equally, if not more important to be able to view the system in terms of simple "task-action models" that link system behavior to the actions a user performs in order to accomplish a set task.

In order to gain access to a rich source of task-action models, we have attempted to link the central organizing image of the Boxer system to geometric intuitions about the way one moves about in space. Structures in Boxer are organized in terms of "boxes." A box is a two-dimensional region in which one can put things, including other boxes. Boxes can thus represent hierarchical structures, and in the interests of integration we use this single structure to organize the following hierarchies, which are treated separately in most programming systems:

- the organization of a user's programs and data according to specific applications (i.e., the kind of organization that is normally supplied by a file system and tree-structured directory)

- the organization of shared meanings for programming language identifiers (i.e., the organization supplied by the scoping rules or block structure of a programming language)

- the organization of how procedures and data are shared among different program modules.

This last organization, above, is closely related to scoping of identifiers, yet distinct from it. It is supplied in Smalltalk, for example, by the class hierarchy, which is a mechanism in addition to Smalltalk's dynamically scoped identifiers. Act-I [7] supplies a similar delegation hierarchy, while maintaining lexical scoping of identifiers. The LISP machine uses dynamically scoped identifiers and supplies the flavor system to deal with this kind of sharing.

Beyond this, boxes incorporate a spatial metaphor to aid in dealing with these organizations. The system user views himself as moving about between boxes (by moving the cursor on the display screen). A box therefore also provides a concrete representation of the context in which the user's input is interpreted. The idea that the entire system can be regarded as a geometric space through which the user moves is, in our opinion, a crucial aspect of making the environment accessible to non-expert users. (This idea was inspired by the Spatial Data Management System developed by the MIT Architecture Machine Group [8]. In Boxer, we extend their spatial metaphor to cover the system's semantics as well as the system's organization of data.)

2.3. No Function in Structure

"No function in structure" is a principle enunciated by John Seely Brown and Johann deKleer as a guide in formulating descriptions of complex systems. Their notion is that when describing a part of a system (e.g., a switch in an electrical

circuit) one should endeavor to phrase the description in terms that are intrinsic to the part itself rather than in terms of the part's function in relation to the entire system. We would like to emphasize a similar principle as a guide in designing complex systems: One should strive to build a system out of "pieces" that can be understood and manipulated without reference to the function they play in the system as a whole. That is to say, there are two ways to build functionality into a system. One is to make many special-purpose parts, each optimized for a particular function. Another is to identify a small number of parts that can be combined to serve many different functions. (This is the difference between a tinker-toy set and a model airplane kit.) The principle says that the system designer should opt for the latter choice.

As an example of how this applies in designing computer systems, consider the issue of "menus." Many systems provide menu-oriented interfaces, in which special symbols on the screen can be selected with a pointing device and executed as commands. This is fine as far as it goes, but suppose the user would like to create his own menu, or edit an existing menu, or write a program that scans the options in a menu. It is all too easy for the system designer to address these needs by implementing menus via a special kind of system object and then providing special-purpose mechanisms for manipulating menus. A better way is to arrange things so that one can realize the functionality of a menu without recourse to special-purpose structures. Thus in Boxer, menu-like behavior is obtained as a natural consequence of the idea that any piece of text on the screen can be selected and executed with "do-it." A menu is simply text (perhaps organized into a box), so that the ordinary operations for manipulating text extend automatically to manipulate menus. In a similar way, one can avoid the need for special-purpose "mail" systems by simply providing the ability to mail any box, regardless of how it was constructed.

2.4. The Importance of Application Examples

This is more a issue of methodology than a specific design principle, but we feel strongly that it is impossible to design a good general purpose programming system without clear images of how the system will be used. Unless this issue is addressed explicitly, system designers will choose their images by default. The problem with this is that it is too easy for the designers to think about the system being used for programming tasks that they themselves will want to accomplish, or worse yet, programming tasks that will want to accomplish in the course of building the system itself. Hence the features that get implemented will be tuned for such "computer science" tasks as implementing a text editor, reducing the size of the kernel system, and so on. At best, these features will be irrelevant to the naive user. More than likely, they will have the detrimental influence of making the system overly complicated.

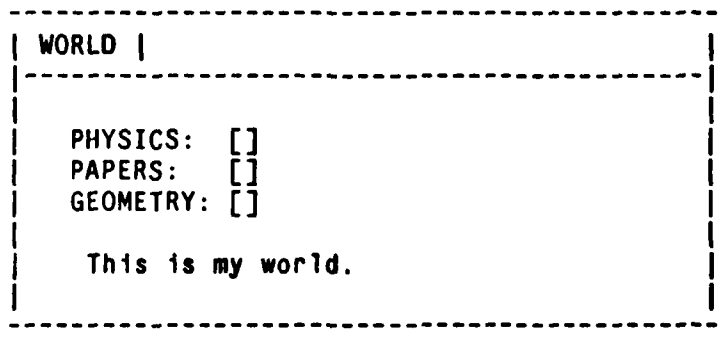
We have tried to avoid this trap in our own work. Accordingly, as we are building the Boxer system from the semantic base upwards, we are at the same time working "from the top down" to formulate detailed examples that illustrate how this highly interactive, graphics-based facility is intended to be used. This entails designing and implementing prototypes of the kinds programs we expect non-expert users to deal with in the Boxer system. These programs are currently implemented in LISP, independently of the Boxer system. As Boxer evolves, we will gauge its flexibility by the ease with which we can incorporate these examples into the integrated computational environment. Section 4 describes some of these programs.

3. A BOXER OVERVIEW

This section sketches a simple scenario that highlights some of the important features of the Boxer system. We caution that although the semantics of the system have been fairly well determined, the details of the syntax and the user interface have still to be worked out, so that the diagrams below should be considered as only rough approximations.

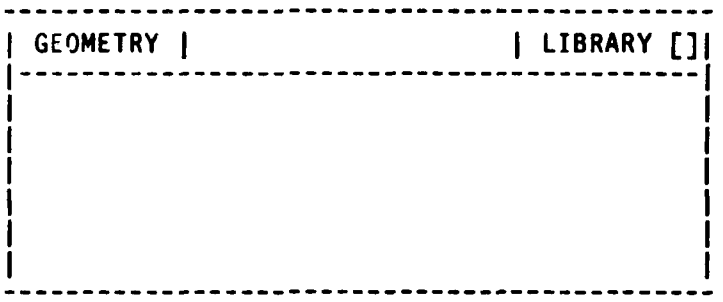
3.1. Basics

The system is structured so that the user's "world" is a hierarchically organized collection of boxes. We see below a sample of the screen as it might appear to us if we enter the system at a fairly high level. Assume that the environment is currently organized into a number of large-scale structures: a PHYSICS box of programs concerned with physics projects, a PAPERS box that contains papers, and a GEOMETRY box that contains programs for doing graphics:



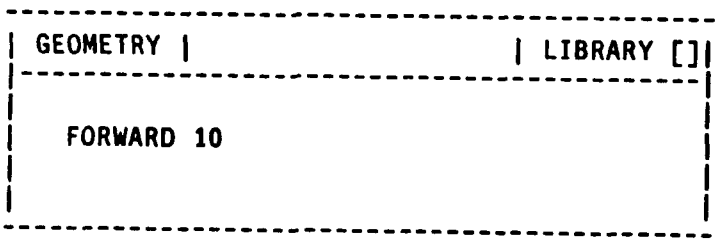
This example shows how boxes can be used to incorporate the functions of a "tree structured directory." The PAPERS box, for instance, is presumably a box of boxes, where each sub-box contains a paper or a collection of papers. Note also that boxes may contain programs or text or (more generally) both.

A major point about boxes is that they function as *environments* through which we can move. Let's suppose we decide to enter the GEOMETRY box. (This is accomplished by moving the cursor to the appropriate box and issuing an "enter" command.) The screen will now appear like this:

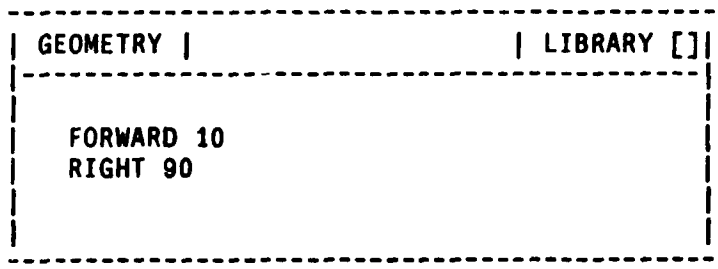


The GEOMETRY box is an environment in which we can write and execute graphics programs. The small box marked LIBRARY in the upper right hand corner is the geometry *local library* that contains definitions of symbols that are local to the GEOMETRY box. In this case they might be built-in procedures for manipulating a graphics cursor, together with any symbols we will define in this environment. We'll assume that the built-in graphics primitives are FORWARD and RIGHT. FORWARD causes a graphics cursor to move forward leaving a trail on the screen. RIGHT causes the cursor to rotate in place. Such a graphics cursor is called a "turtle." Drawing pictures by moving a cursor with FORWARD and RIGHT commands leads to a new approach to the study of geometry, called "Turtle Geometry" [1].

We can now type text to be edited and/or executed. For example, typing "FORWARD 10":

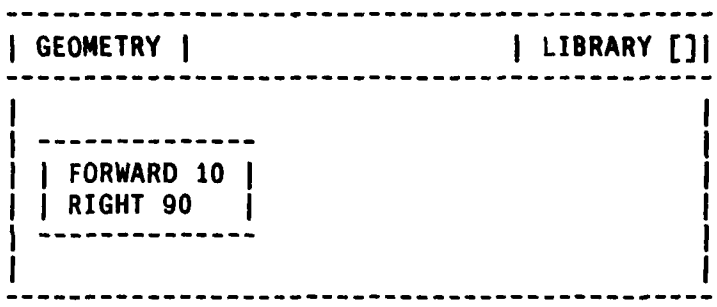


followed by "do-it" will make the turtle cursor move forward and draw a line 10 units long. Once the text of the FORWARD command is on the screen, it can be re-executed any number of times by pointing to it and specifying "do-it." It can also be followed by another command, such as

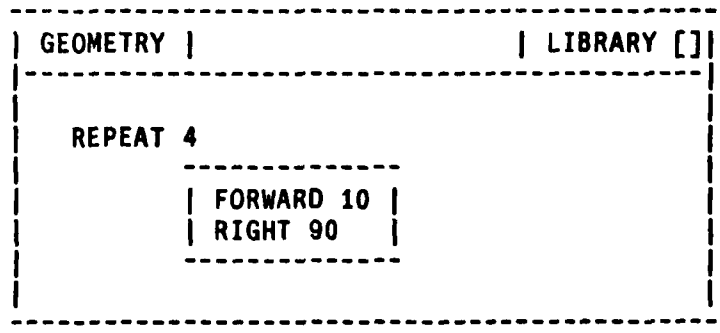


to make the turtle rotate right 90 degrees. If we leave both commands on the screen, then we automatically obtain the functionality of a menu for issuing graphics commands. In fact, the geometry box could have been stored like this to begin with, so that a user entering the geometry environment could automatically obtain such a menu.

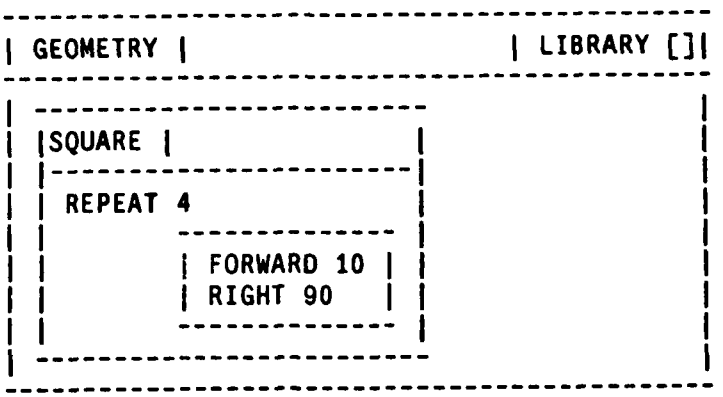
We can also group the commands by drawing a box around them:



Pointing to this smaller box and indicating "do-it" makes the turtle draw a right-angle corner. Doing this four times:



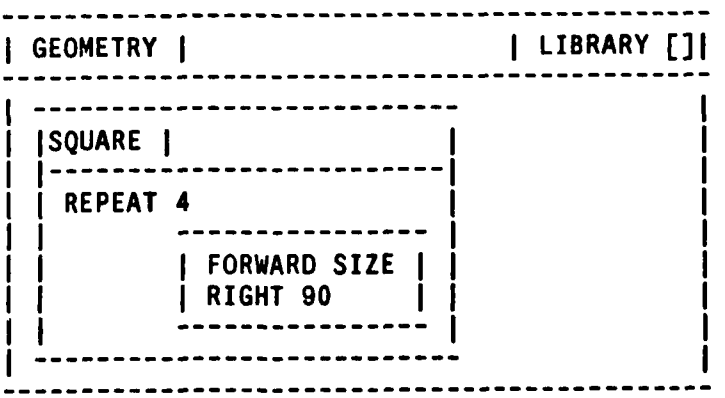
makes the turtle draw a square. (This shows how boxes can incorporate the syntactic grouping functions of the "begin -- end" blocks of Algol-like languages.) If we like, we can draw a box around this new sequence, and name it SQUARE:



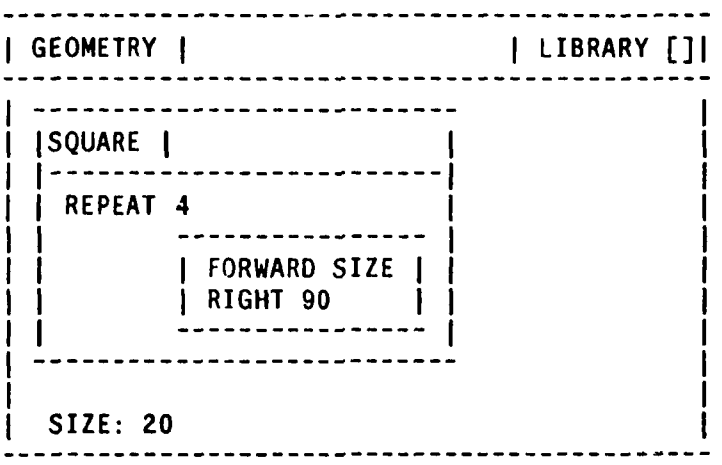
Naming the box effectively defines SQUARE as a procedure, and this definition is installed in the GEOMETRY local library. SQUARE can now be used as a procedure within the GEOMETRY box. This is a very simple example of the "concrete programming" style mentioned in Section 2.1.

3.2. Scope of Variables

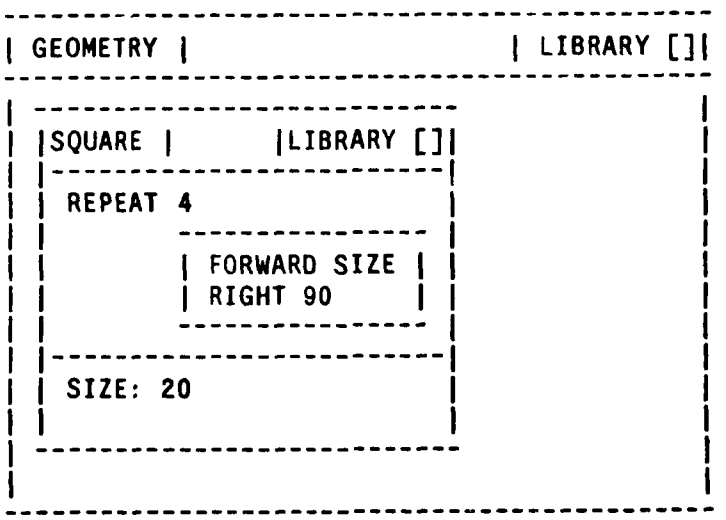
The nesting of the boxes also determines the nesting of environments as far as the scoping of variables is concerned. Identifiers are searched for first within the local library of the box in which the code is being executed, then in the local library of the containing box, and so on. For instance, we could edit the SQUARE procedure so that the size of the squares drawn will be determined by a variable SIZE:



We can set SIZE in the GEOMETRY environment, by moving the cursor within the GEOMETRY box and assigning a value to it:



Then executing SQUARE within the GEOMETRY box will draw squares of size 20. Alternatively, we could define SIZE to be a symbol *local* to the SQUARE procedure by placing the symbol in SQUARE's local library and executing the assignment within the SQUARE box:

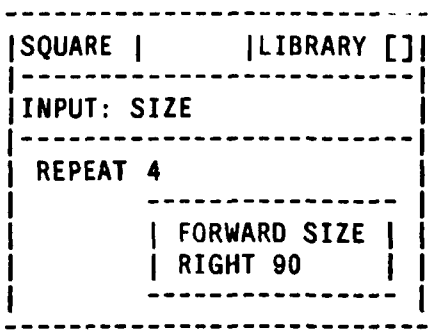


Rather than executing the assignment from within SQUARE, we could also cause the same effect from outside the SQUARE box by using the TELL command, as in

```
TELL  SQUARE  [SIZE:  20]
```

In general, TELL means "execute the following command within the designated box." Notice that the "set variable and execute code" style we have been using is an intermediate step between defining only fixed-instruction procedures, and defining procedures with parameters. This allows beginning users to realize some of the benefits of parameterized procedures without having to master the parameter concept.

If we would like SIZE to be a parameter for SQUARE we can specify this as follows:

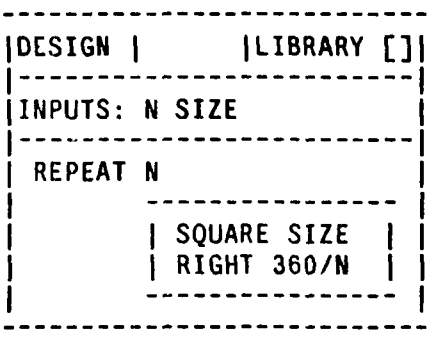


The INPUT section of the box informs the system that SQUARE expects an input when it is executed as a procedure, e.g.,

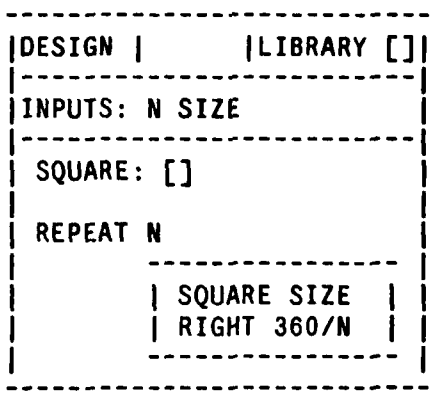
```
SQUARE  20
```

The input to SQUARE is evaluated and stored under the name SIZE in the box's local library. If the input is missing, the system can use the INPUT information to generate an appropriate prompt.

Naturally, SQUARE can be used in defining new procedures, e.g.,



Here DESIGN will be able to call SQUARE, presumably because DESIGN will be run in the GEOMETRY environment in which SQUARE is defined. It is also possible to make SQUARE a *local procedure* to DESIGN, simply by including the SQUARE box within the DESIGN box:



(Notice that the graphics interface allows the user to suppress the details of sub-boxes like SQUARE within DESIGN.) SQUARE will now be included in the DESIGN local library. This shows how the hierarchical structure of boxes enables one to obtain all the advantages of a block-structured language. But a major difference between Boxer and most block-structured languages is that the blocks are *interactive*, that is, the user can enter and leave blocks, modify blocks, define new local programs and variables, and so on. In essence, the blocks are used to structure not only the static program organization, but also the user's interaction with the system.

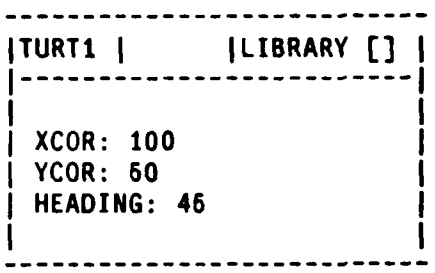
Here is a summary of how Boxer interpretation can be viewed in an "Algol Contour Model-like" manner. Identifiers are searched for in the succession of local libraries, starting with the box where reference to the identifier is made, and working outward. When a procedure is called, the box which is the procedure is copied into the *calling environment*, inputs are entered into the box's local library and the code portion of the box is executed. Incidentally, this shows how the "box" representation automatically incorporates a contour model that can be used to explain the semantics of the system.

3.3. Object-Oriented Programming

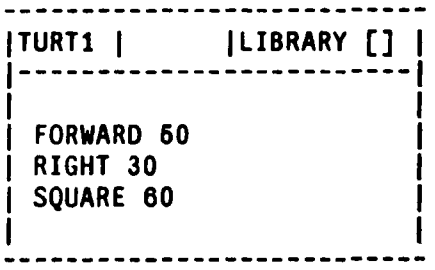
Besides serving as procedures and environments, boxes also can be used to program in an object-oriented, or "message-passing" style. As an example, we'll implement a turtle as an object, and show how to build a system with multiple turtles.

To keep things simple, suppose that the only state variables relevant to a turtle are its heading and the x and y coordinates of its position on the display screen. We'll assume that the commands FORWARD and RIGHT (in the GEOMETRY local library) have been implemented so as to refer to free variables XCOR, YCOR, and HEADING.

We can implement a turtle, say TURT1, as a box containing these variables in its local library, for example, by building a box in which the variables are assigned appropriate initial values:



To make TURT1 go FORWARD, turn, or draw a square, we simply execute the appropriate commands from within this box:



Notice that the dynamic scoping discipline determines that the XCOR, YCOR, and HEADING variables accessed by FORWARD and RIGHT will be the ones local to TURT1. We could also make TURT1 move by using the TELL command from outside the box, as in

```
TELL  TURT1  [FORWARD  50]
```

This provides the capability that would be expressed in actor languages as "sending the object HURT1 the message FORWARD 50."

Of course, we could have another turtle, TURT2 that is identical to TURT1 except for its name, and could give commands to them either by entering the appropriate turtle box and typing the command, or by using TELL, for example, as in:

```
+-------------------------+
|GEOMETRY|    |LIBRARY [] |
|-------------------------|
| TURT1: []               |
| TURT2: []               |
|-------------------------|
|                         |
| TELL TURT1 [FORWARD 10] |
| TELL TURT2 [SQUARE]     |
|                         |
+-------------------------+
```

Notice that TURT1 and TURT2 share knowledge about FORWARD, RIGHT and SQUARE by virtue of their inclusion in the GEOMETRY box.

Continuing, suppose we wanted TURT2 to execute FORWARD commands in some special way. We could do this by including the special FORWARD as a local procedure within TURT2:

```
+-------------------------+
|TURT2 |      |LIBRARY [] |
|-------------------------|
|                         |
| FORWARD: []             |
|                         |
+-------------------------+
```

Then whenever we entered the TURT2 box, or used TELL TURT2, we would obtain this local FORWARD procedure. Moreover, in a situation such as

```
+-------------------------+
|GEOMETRY|    |LIBRARY [] |
|-------------------------|
| TURT1: []               |
| TURT2: []               |
|-------------------------|
|                         |
| TELL TURT1 [SQUARE]     |
| TELL TURT2 [SQUARE]     |
|                         |
+-------------------------+
```

we would have TURT1 executing SQUARE using the default FORWARD procedure (i.e., the one contained in the GEOMETRY local library) and TURT2 executing SQUARE using its local FORWARD procedure. This example shows how the hierarchical box structure can be used to capture the "class-subclass" structure of actor languages.

4. SAMPLE PROGRAMS FOR EDUCATIONAL USE

We stressed in Section 2.3 the importance of clear images to guide the development of any system aimed at non-expert users. This section presents three examples. All are implemented in LISP and (in various stages of completion) are currently running on the LISP machine. Over the next few months we will continue to refine the LISP implementations in response to observations of non-programmers using these tools.

4.1. Vector Fields

David Andre has designed and implemented a system for exploring two-dimensional vector fields. Figures 4-1 and 4-2 (a through f) show snapshots of a typical interaction with the system. In (a) we see a menu of "top level" choices. These include creating a new vector field, displaying a previously created field, zooming in or out on a portion of a displayed field, dropping a particle into a field, creating a new kind of particle, and so on. Here the user has selected (by using the mouse) to create a new field. In frame (b) the system prompts with another menu giving the kinds of fields it currently knows about: inverse square, inverse linear, and constant. (New kinds of fields are easily added by specifying them as vector-valued functions of position.) The user selects an inverse square field, and the system responds by prompting for the magnitude of the source and asking that the pointer be moved to indicate the position of the source. The user now selects "done," and the resulting inverse square field is as shown in frame (c). More generally, the user could have selected another field from the options menu, and this would be be superposed with the currently specified inverse square field. In this way, one can easily build up complex fields as superpositions of the basic types.

Frame (c) also shows the user selecting the "create particle" option. Choices to be specified for a particle include the its shape as displayed on the screen, whether it leaves a trail as it moves, and how it interacts with the field (e.g., as a charged particle, a direction follower, or in some other way that is easily specified by inputting a procedure that computes the particle's new velocity as a function of old velocity and field components). In this example, the user creates a charged particle and drops it into the field. In frame (d) the system prompts for the particle's initial position and velocity. The resulting path is shown in frame (e). Frame (f) shows another example--a charged particle dropped into a dipole field (constructed as the superposition of two inverse square fields).

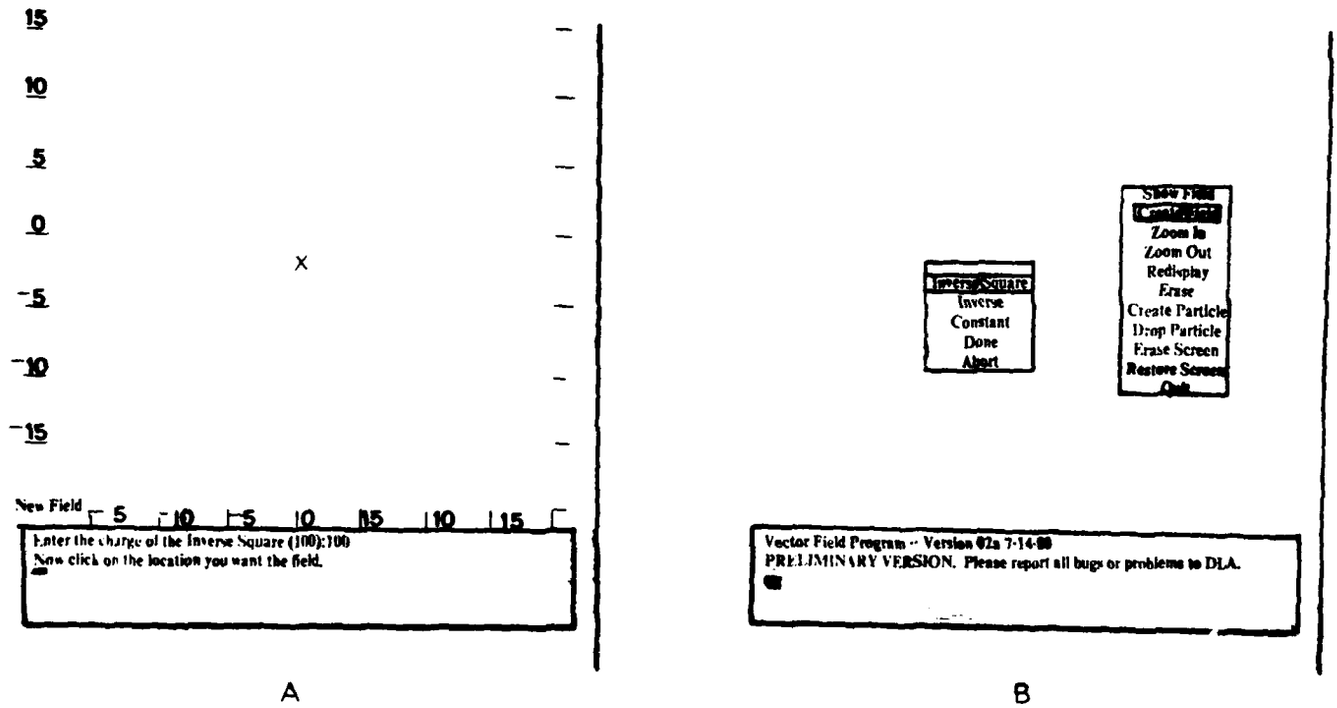


Figure 4-1: Vector Field Program (a) and (b)

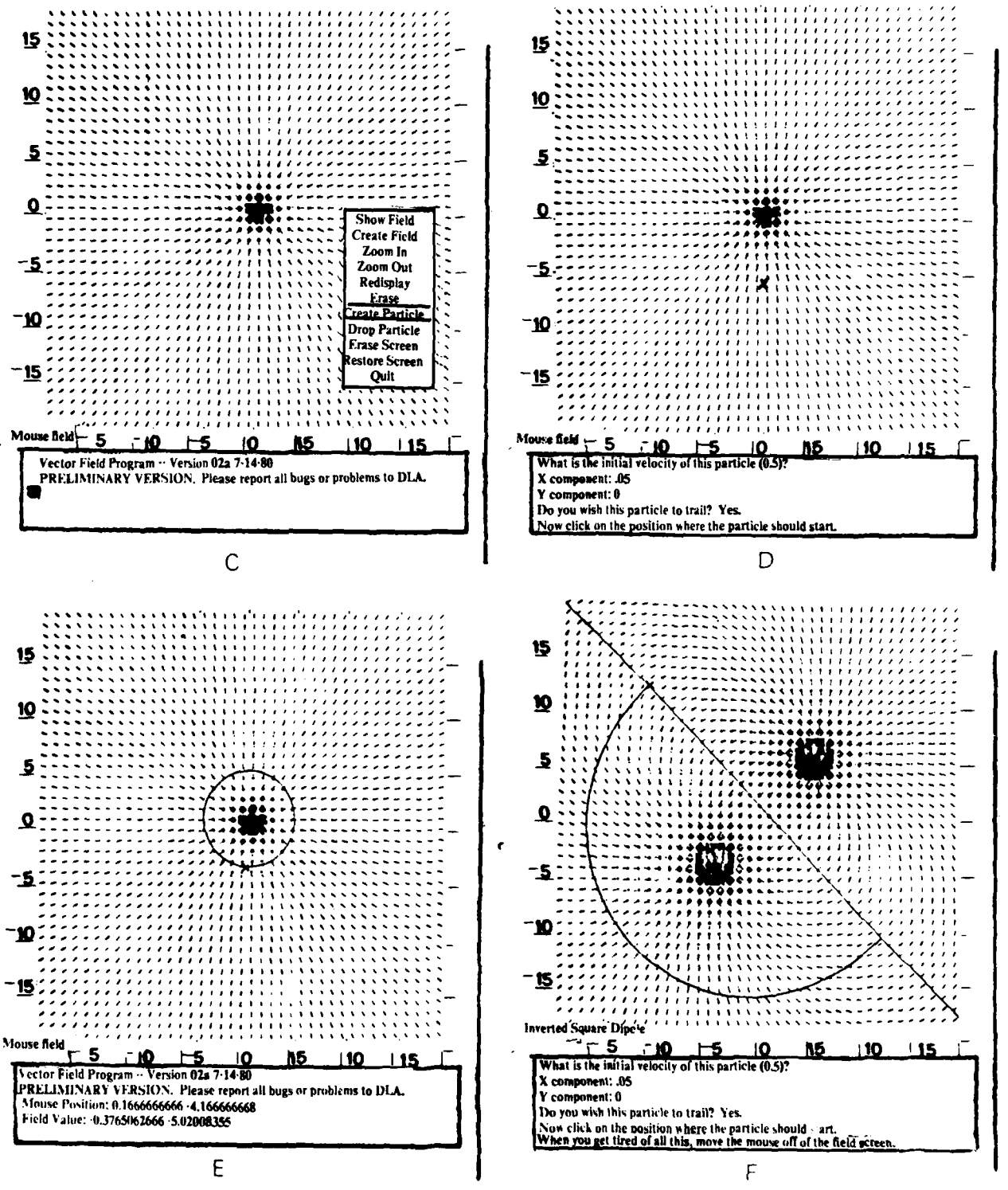


Figure 4-2: Vector Field Program (c) through (f)

4.2. Deformations of Structures

The "stress machine" (being implemented by Gregor Kiczales) is a tool for exploring how structures deform under mechanical stress. In our experimental teaching sessions, we will be using this to investigate the possibility of concentrating on momentum flow as an alternative perspective to "F = ma" in the presentation of elementary mechanics [9].

The program enables one to build structures out of nodes connected by elastic bars. There are also "forces" (nodes which exert a constant force in some direction) and "supports" (nodes with fixed position). Frames (a) and (b) of Figure 4-3 show the user making a structure consisting of two supports and a (vertically downwards) force. In frame (c), the user requests to change some of the attributes of the force. This can be accomplished by pointing at the current value of the attribute and typing in a new value. In this case, the user has added a horizontal component to the force. (Note the change in the direction of the force arrow in going from (b) to (c).) Now the user activates the structure, and frame (d) shows how it deforms under the indicated stress. Frames (e) and (f) show how a more complex "bridge" structure deforms in response to a load.

4.3. Interacting Particles

James Dempsey's "particle" program is a tool for exploring the dynamics of many-particle systems. The program enables the user to create barriers and containers, to fill these with particles, to specify different kinds of particle interactions, and to accumulate statistics about the particles' behavior, such as the average temperature or pressure. We plan to use this program in teaching statistical mechanics to undergraduates.

Frame (a) of Figure 4-4 shows a pointer being used to construct the walls of a box to be filled with particles as shown in frame (b). In frame (c), the user specifies that subwindows 1 and 2 are to show respectively, a graph of pressure versus time, and a histogram of the temperature (particle energy) distribution. The system is now told to "go" and the information evolves as shown in frame (d). In frame (e) the user directs that the bottom wall of the box is to be heated, so that particles bouncing against it will pick up extra energy. One now sees (frame f) how this affects the evolving pressure and temperature statistics. Frames (g) and (h) show particles escaping through a hole in a container. Further improvements to the system will allow the user to set up "Maxwell's demons" in such situations and to measure their impact on entropy.

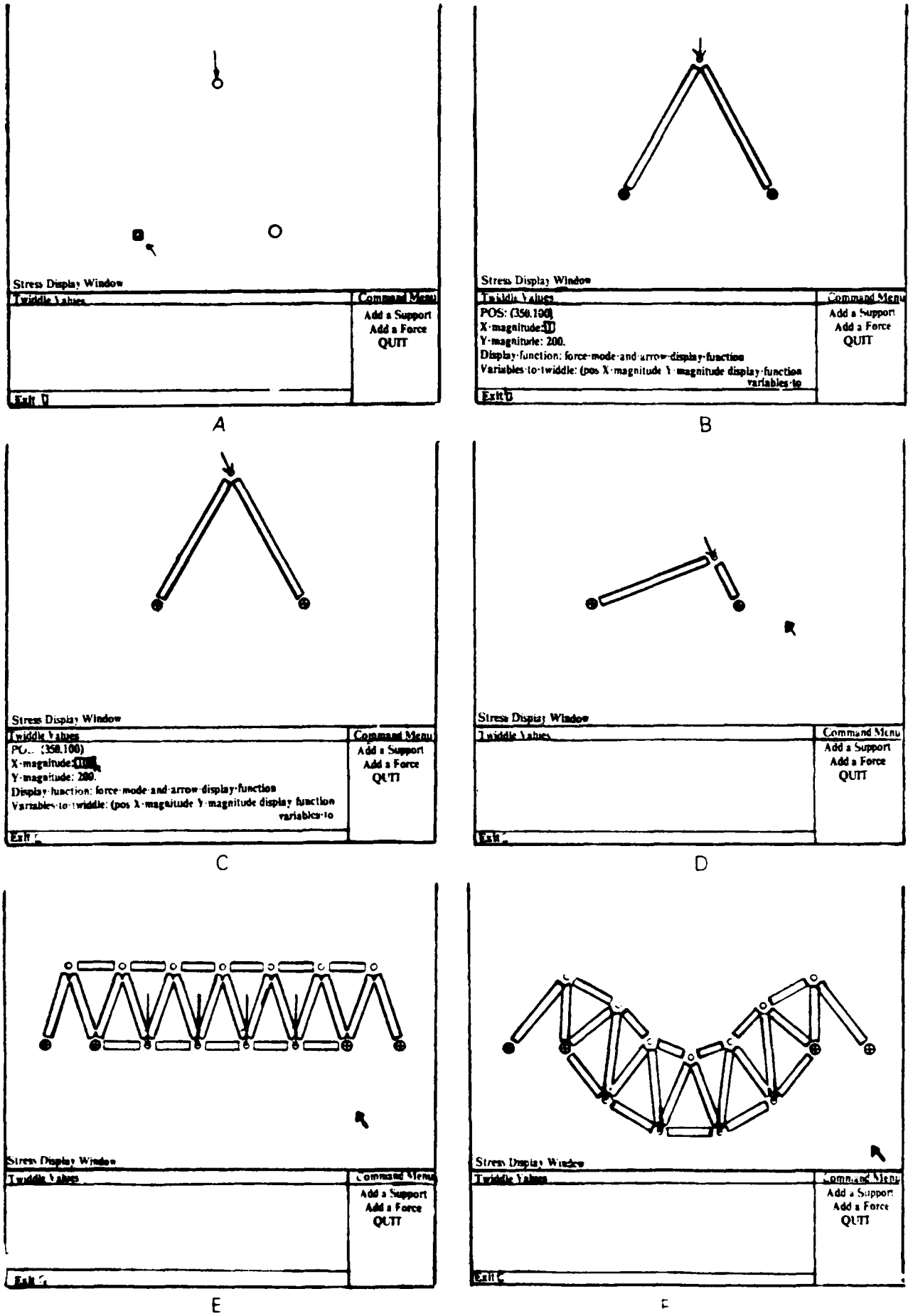


Figure 4-3: Stress Machine (a) through (f)

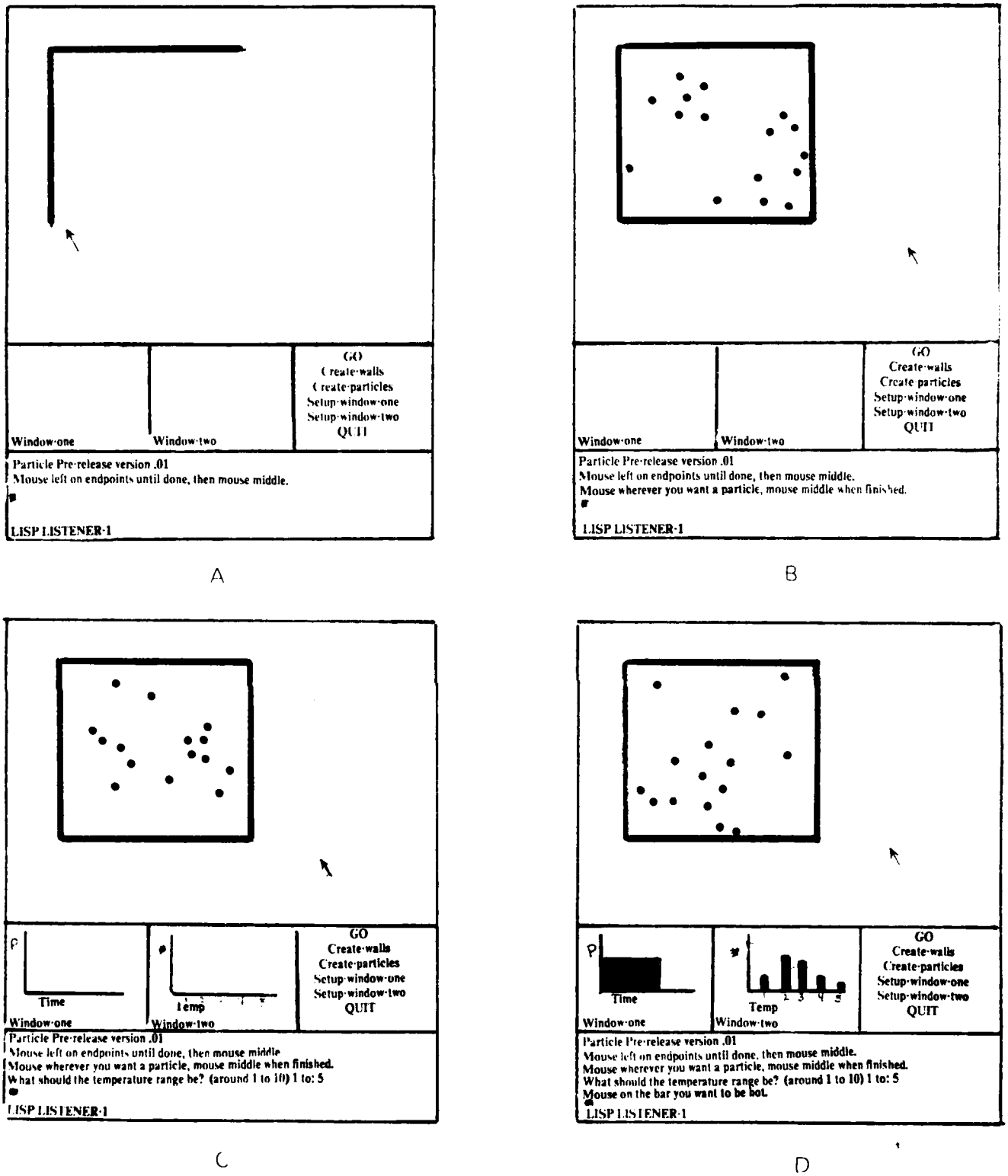
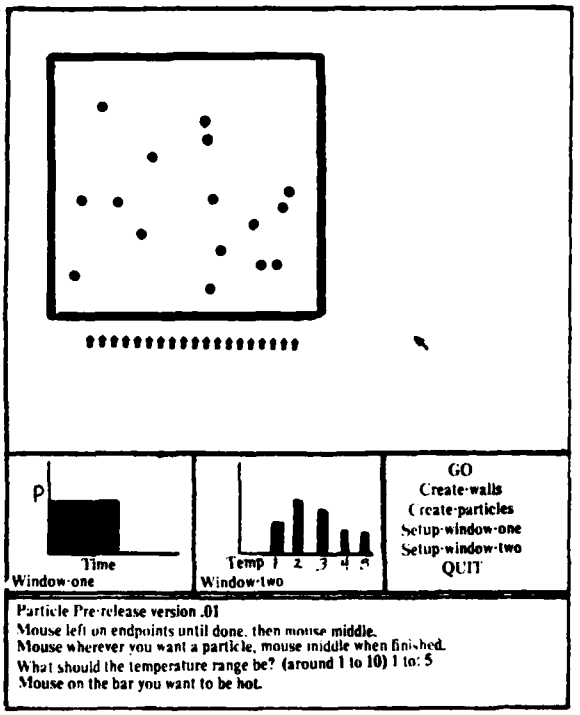
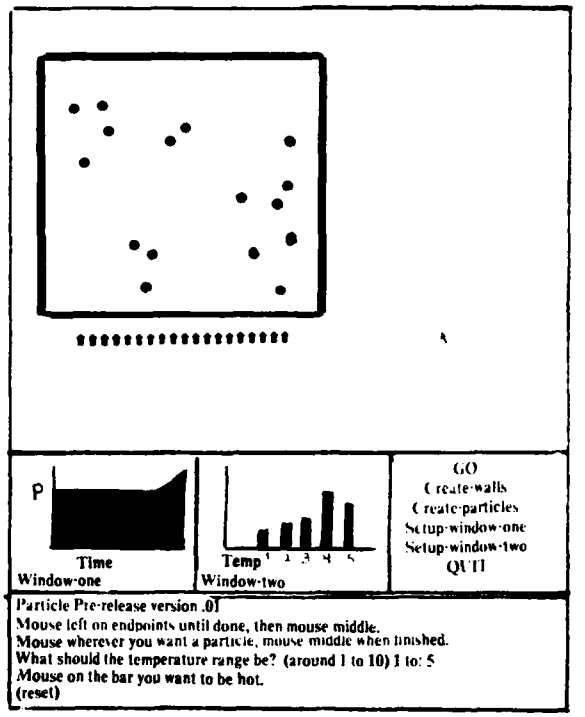


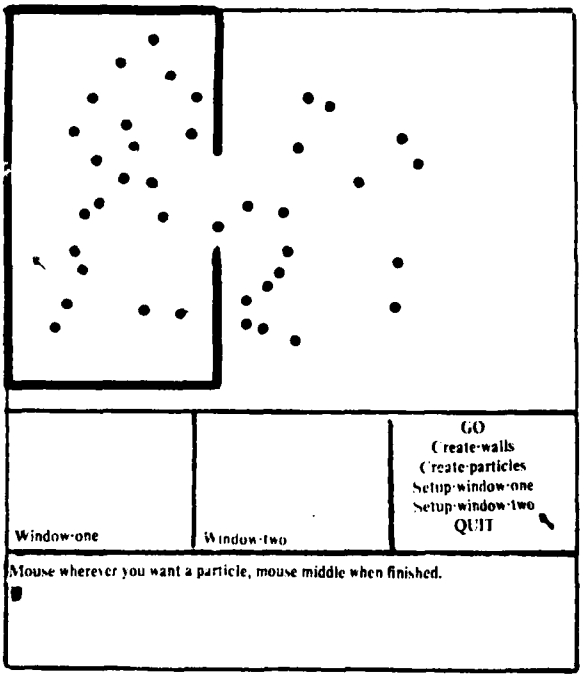
Figure 4-4: Particle system (a) through (d)



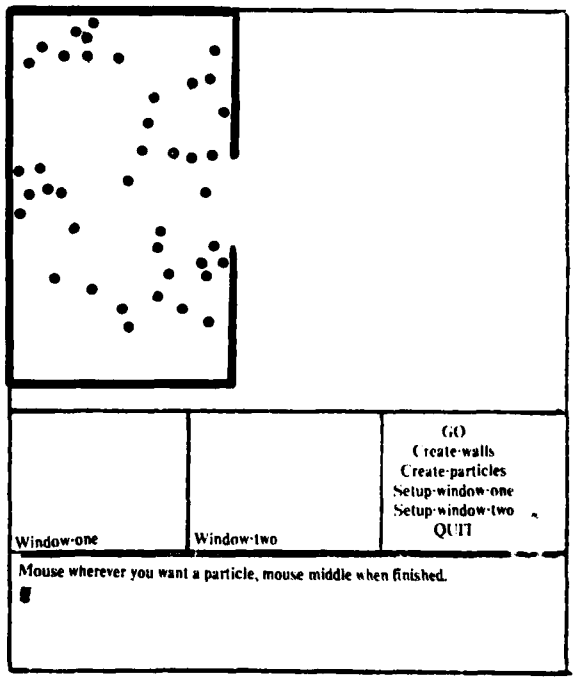
E



F



G



H

Figure 4-5: Particle system (e) through (h)

5. NEXT STEPS

Our obvious next step is to begin implementing the graphics interface. Combined with the interpreter, this will produce a version of the system as described in Section 3. We hope to complete thus over the summer. At the same time we will attempt to translate some of the sample programs of Section 2.4 into the Boxer language. At this point we will have a usable prototype system running on the LISP machine. We would then like to conduct some experiments in which we observe novice users interacting with the system, and to assemble psychological data to inform the next round of system development. We also expect to investigate a number of enhancements to the basic system, including:

- Automatically generating help and documentation facilities. Possibilities range from simple command completion, to automatic prompting for function inputs, to changing the nature of the user interaction so that programs are normally generated by the user filling in templates rather than by simply typing text.

- Extending the "concrete programming" facilities by incorporating some of the ideas in Lieberman's Tinker system.

- Allowing for "declarative mode programming," in which the user specifies relationships that the system should maintain automatically (as done, for example, in Alan Borning's "Thinglab" System.[10])

We expect to continue this kind of development for about a year, working with the LISP machine. Hopefully, the situation concerning personal computing hardware possibilities for LCS will be clear enough by that time so that we can settle on a configuration for large-scale implementation.

References

1. Abelson, H. and diSessa, A. Turtle Geometry: The Computer as a Medium for Exploring Mathematics, Cambridge, MA, MIT Press, 1981.

2. Dertouzos, M.L. et. al Proposal for Continuation of Research, January 1, 1981 - December 31, 1982, submitted to the Defense Advanced Research Projects Agency, MIT Laboratory for Computer Science, Cambridge, MA, August 1980.

3. Smith, D. "PYGMALION: A Creative Programming Environment," Ph.d. dissertation, Stanford AIM 260, Stanford University, Stanford, CA, June 1975.

4. Lieberman, H. and Hewitt, C. "A Session with Tinker: Interleaving Program Testing with Program Design," MIT Artificial Intelligence Laboratory Memo 577, Cambridge, MA, September 1980.

5. Goldstein, I. "PIE: A Network-Based Personal Information Environment," Proceedings of the Office Semantics Workshop, Chatham, MA, June 1980.

6. Young, R. "User Models of Hand Calculators," to appear in Journal of Man-Machine Interface, 1981.

7. Lieberman, H. "Sharing Knowledge by Delegating Messages," MIT Artificial Intelligence Laboratory, Cambridge, MA, 1979.

8. Bolt, R.A. Spatial-Data-Management, Massachusetts Institute of Technology, Cambridge, MA, 1979.

9. diSessa, A. "Momentum Flow as an Alternative Perspective in Elementary Mechanics," American Journal of Physics, 48 (1980), 365-369.

10. Borning, A. "ThingLab--An Object-Oriented System for Building Simulations Using Constraints," Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

Publications

1. Abelson, H. and diSessa, A. Turtle Geometry: The Computer as a Medium for Exploring Mathematics, Cambridge, MA, MIT Press, 1981.

2. diSessa, A. "Momentum Flow as an Alternative Perspective in Elementary Mechanics," American Journal of Physics, 48, (1980), 365-369.

3. diSessa, A. "Computation as a Physical and Intellectual Environment in Learning Physics," Computers and Education, 4, 1, (1980), 67-75.

4. diSessa, A. "An Elementary Formalism for General Relativity," American Journal of Physics, (May 1981)

5. diSessa, A. "The Computer and Mathematical Experience," to appear in Proceedings of the Fourth International Congress on Mathematics Education, Berkeley, CA, August 1980.

6. diSessa, A. "Unlearning Aristotelian Physics: A Study of Knowledge--Based Learning," to appear in Cognitive Science, 1981.

7. diSessa, A. "Phenomenology and the Evolution of Intuitions," to appear in Mental Models, D. Gentner and A. Stevens, (Ed.) Lawrence Erlbaum Press, 1980.

Talks

1. Abelson, H. "Computational Environments for Non-Expert Users," Institut fuer Informatik, Universitaet Stuttgart, December 1980.

2. Abelson, H. "Using Computers in Mathematics Education," Institut fuer Didaktik der Mathematik, Universitaet Bielefeld, December 1980.

3. Abelson, H. "Designing Educational Computing Environments: Putting the User in Charge," Workshop on Computer Science and Cognitive Science, Kaelberbronn, West Germany, April 1981.

4. diSessa, A. "Computers and Mathematical Experience," Fourth International Congress on Mathematical Education, Berkeley, CA, August 1980.

5. diSessa, A. "Phenomenology and the Evolution of Institutions," Conference on Mental Models, La Jolla, CA, October 1980.

6. diSessa, A. "The Effect of the Computer in Changing the Content of Instruction," New York Academy of Sciences, NY, December 1980.

7. diSessa, A. "Knowing How to Teach is Also Knowing What to Teach," Annual Meeting of the American Association for the Advancement of Science, Toronto, Canada, January 1981.

8. diSessa, A. "The Role of Experience in Learning," International Conference on Cognitive Science and Computer Science, Freudenstadt, W. Germany, March/April 1981.

9. diSessa, A. "The Turtle that Teaches Physics,"

   Bank Street College of Education, NY, April 1981;
   New York Academy of Sciences, NY, April 1981.

FUNCTIONAL LANGUAGES AND ARCHITECTURE

Academic Staff

Arvind, Group Leader

Research Staff

R.E. Thomas

Graduate Students

T. Chu
V.Kathail
K. Pingali

Undergraduate Students

S. Heller
D. Perich
J. Rodriguez
E. Seidewitz
K. Suh

Support Staff

K. Warren

Visitor

T. Shimada

Functional Languages and Architecture

1. INTRODUCTION

The Functional Languages and Architectures Group was formed in January 1981 to explore new computer structures which can effectively exploit the parallelism evident in many functional programs. The following summarizes the work done since the group's formation as well as the time when we were part of the Computation Structures Group.

Our approach in studying effective exploitation of parallelism is based on a highly dynamic interpreter for dataflow graphs called the U- interpreter . Since we believe the success of a general- purpose multiprocessor computer depends on its effective programmability and its efficient utilization of resources, we are concerned not only with hardware issues but also with associated system problems such as high level language support, communication requirements, and efficient distribution of workload over the machine. As one step in addressing these issues we have augmented the high level dataflow language Id with a new data type called I-structure which can be used for efficient programming of numerical algorithms dealing with arrays [1]. On the architecture side we have completed the functional specification of a machine which provides hard ware support for I-structures and dynamic procedure invocation. The main factors which influenced our design were the possibility of using custom chips and the insight into large scale problems gained by our study of partial differential equation programs [2]. We have also studied a general scheme to decompose a functional program into smaller segments such that each segment can be executed on a single processor to minimize interprocessor communication. At this stage we feel the best way to test our ideas is by constructing a prototype dataflow computer.

2. LANGUAGE RELATED WORK

As part of his master's thesis, K. Pingali is studying the problem of implementing streams and managers which are necessary for dealing with resource management and input/output. Since a stream can be potentially infinite, reasonable implementation seems to require a *demand driven approach*. This led Mr. Pingali to study the tradeoffs between *purely data driven* and*purely demand driven* approaches. In the case of infinite computations, a purely data driven approach may generate activities which do not contribute to the final result. These activities may spread over the dataflow machine and uselessly consume resources. However, a purely demand driven approach may not be able to exploit parallelism as much as a

purely data driven approach; it also requires twice the amount of communication because one demand must be sent for each data token generated. We are convinced that in the presence of potentially infinite computations a mixed approach is best. Finite computations can then be data driven whereas obviously infinite computations could be demand driven. Moreover even infinite computations need not have demands propagate node by node throughout the graph. The node where a demand ultimately terminates can often be identified at compile time. Suitable code can be generated which transmits demands directly to these nodes which can then initiate the rest of the computation according to the data driven approach. In this way the number of demand arcs in the graph can be substantially reduced.

As part of our ongoing language work, V. Kathail and K. Pingali have changed the syntax of Id to facilitate parsing and to support compile time determination of the arity of expressions and independently compiled procedures. We have completed a new parser for Id which will be shared among at least two compilers - one to translate Id (including streams) to MACLISP and one too generate dataflow graphs for the prototype dataflow computer.

3. A DATAFLOW ARCHITECTURE AND PROTOTYPE IMPLEMENTATION

We have completed the functional specification of a dataflow machine based on the U-interpreter. It is designed so that any functional language which can be translated into our graphical base language can be executed on our machine. The machine consists of N processing elements communicating via a bit-serial packet switched communication network. Each processing element (PE) in our specification contains a floating point ALU, 64K of program storage, 64K of I-structure storage, and special hardware to deal with matching and manipulation of tags. The communication system routes fixed-size packets from one PE to another PE. Each packet (token) carries a data value and a fixed-size tag that indicates the name of the activity to which the token is destined. A tag contains the following four fields:

- *PE-number* - the address of the processing element to which this token is destined;

- *color* - the procedure activation to which this token is destined;

- *s* - the physical address in the memory where the operation to by executed is stored;

- *i* - the iteration count if the activity is inside a loop. Activities outside loops have an iteration count of 1.

The basic operation of the machine is to bring together tokens which have identical tags, execute the desired operation, and generate one or more tokens each holding a result value along with the tag of the destination activity. Activities are distributed over the PE's according to one or more hashing functions which map from tags to PE numbers.

We plan eventually to implement our design with custom VLSI chips and commercially available memory boards. However, we have decided to first construct a prototype machine which emulates all required functions using two M68000 microprocessors per PE. R. Thomas is in charge of developing the overall strategy, design, and implementation of the prototype. The goals of this project are: to identify those modules which benefit most from reimplementation in custom hardware before commitment is made to extensive hardware design, and to construct and interconnect up to 256 PE's to test whether our machine will indeed show significantly higher performance when large numbers of PE's are available.

At this time we are planning to use forty-bit tags in the prototype · bits for PE number, four for color, sixteen for s, eight for i, and four for system use. The validity of these sizes is to be determined by hardware and software simulations. Further details of the architecture can be found in [3][4]. However, there are several features of the architecture which are worth mentioning as they distinguish our work from other proposed dataflow architectures such as [5][6][7][8] [9][10].

3.1. Distribution of Activities over PE's

For efficient resource utilization it is sometimes necessary to distribute uniformly the activities in the machine over time and space. We have come to the conclusion that a fully dynamic distribution of activities (i.e., every activity is individually assigned at run time) is inefficient. We have therefore developed a scheme for activity distribution which operates in two steps:

1) A group of PE's, known as *physical domain*, is allocated for each invocation of a loop or procedure. All activities of a loop or procedure take place within that physical domain.

2) The activities of a procedure or loop are distributed over the PE's in a physical domain using a simple static mapping (i.e., hashing function) based on the s part, the i part, or both parts of a tag.

Once a physical domain and a mapping scheme have been selected, the code needed by the PE's in that domain is fixed and can be preloaded in their program memories. Further, if we assume that program graphs are stored using forward

links, then static relocation will eliminate the need for dynamically evaluating the mapping function.

The scheme for mapping programs discussed above needs a scheduler to allocate physical domains. The scheduler is called when a procedure (or loop) is invoked and it selects a domain by taking into account factors such as how many activities are expected to be generated, the size of the code block, whether the code block is already present in some other physical domain, and how much data has to be moved between the invoking physical domain and the new physical domain. It appears at this stage tha. the scheduler would need hints from the user to perform its task optimally. The scheduler could be a program executed by a predesignated PE or PE's, or alternatively it could be a special purpose processor. T. Shimada has developed a functional simulator in MACLISP for our machine which we expect to be helpful in scheduler behavior.

3.2. Reusable Fixed-Size Tags

A tag carried by a token in our machine is a hardware implementation of an abstract entity called an activity name as defined by the U-interpreter . Activity names may become arbitrarily large as a result of recursive procedure calls. However, a tag which is interpreted by our hardware design is fixed in size and thus we have developed a scheme to represent activity names with fixed-size reusable tags. Like processors and memory, a group of tags (i.e., tags with the same *color* field) is allocated and deallocated for each procedure or loop invocation. The scheduler which allocates the physical domain to a procedure invocation is also responsible for allocating its new color.

3.3. I-structure Storage and System Generated Tokens

Associated with each PE is 64K bytes of I-structure storage which is part of a single address space in the machine. Our architecture allows the elements of a single I-structure to be distributed over several PE's. The distribution uses a mapping scheme which ensures that in many cases an element of an I-structure needed by a PE is available in that PE's own I-structure storage. However, in certain cases I-structure operations involve sending a request to some other PE to store or retrieve values. Other examples of similar communication requirements are a request from the scheduler to set a register in a PE or a request to read the status of a PE for diagnostic purposes. To accomplish this kind of communication we have incorporated system tokens which are distinguished from ordinary tokens by a one bit field. System tokens also carry an opcode specifying the nature of the request on the token itself.

3.4. Implementation of Procedures

Most functional languages require efficient support for dynamic procedure invocation to achieve programming flexibility. However, among the proposed dataflow machines, few support procedures in any generality. Our machine provides for a variety of procedure calling conventions. For example, using compiler generated triggers to start the execution of an activity it is possible to implement either strict or nonstrict procedure call semantics [3].

References

1. Arvind and Bryant, R.E. "Design Considerations for a Partial Differential Equation Machine," Scientific Computer Information Exchange Meeting, September 1979, 94-102.

2. Arvind, Kathail, V. and Pingali, K. "A Dataflow Architecture with Tagged Tokens," MIT/LCS/TM-174, MIT Laboratory for Computer Science, Cambridge, MA, September 1980.

3. Arvind, Gostelow, K.P. and Plouffe, W. "An Asynchronous Programming Language and Computing Machine," Technical Report 114a, University of California Department of Information and Computer Science, Irvine, CA, December 1978.

4. Arvind and Kathail, V. "A Multiple Processor Dataflow Machine that Supports Generalized Procedures," Computation Structures Group Memo 205, MIT Laboratory for Computer Science, Cambridge, MA, February 1981.

5. Arvind and Thomas, R.E. "I-Structures: An Efficient Data Type for Functional Languages," MIT/LCS/TM-178, MIT Laboratory for Computer Science, Cambridge, MA, September 1980.

6. Amamiya, M. et al. "Dataflow Machine Architecture," Internal Notes 1 and 2, Musashino Electrical Communication Laboratory, Nippon Telephone and Telegraph, Japan, May 1980.

7. Davis, A.L. "The Architecture and System Methodology of DDM1: A Recursively Structured Data Driven Machine," IEEE Proceedings of the 1978 International Conference on Parallel Processing, April 1978, 210-215.

8. Dennis, J.B., Boughton, G.A. and Leung, C. "Building Blocks for Data Flow Prototypes," Proceedings of the 7th Annual Symposium on Computer Architecture, May 1980, 1-8.

9. Gostelow, K.P. and Thomas, R.E. "Performance on a Simulated Dataflow Computer," IEEE Transactions on Computers, C-29, 10, (October 1980), 905-919.

10. Johnson, D. et al. "Automatic Partitioning of Programs in Multiprocessor Systems," COMPCON Spring 80, February 1980, 175-178.

Publications

1. Arvind "Decomposing a Program for a Multiple Processor Machine," Proceedings of 1980 International Conference on Parallel Processing, Bayne Highlands, MI, August 1980.

2. Arvind and Gostelow, K.P. "The U-Interpreter," to appear in Computer, 1981.

3. Arvind, Kathail, V. and Pingali, K. "A Processing Element for a Large Multiple Processor Dataflow Machine," Proceedings of the IEEE International Conference on Circuits and Computers, 2, 1980, 601-605.

4. Arvind and Thomas, R.E. "I-Structures: An Efficient Data Type for Functional Languages," MIT/LCS/TM-178, MIT Laboratory for Computer Science, Cambridge, MA, June 1980.

5. Arvind, Kathail, V. and Pingali, K. "A Dataflow Architecture with Tagged Tokens," MIT/LCS/TM-174, MIT Laboratory for Computer Science, Cambridge, MA, September 1980.

6. Arvind and Kathail, V. "A Multiple Processor Dataflow Machine that Supports Generalized Procedures," Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, MN, May 1981.

Theses in Progress

1. Pingali, K. "Streams and Dataflow: Implementation Issues," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1981.

Talks

1. Arvind "A General Purpose U-Interpreter Based Dataflow Architecture," IBM Research Center, Yorktown Heights, NY, July 1980.

2. Arvind "Decomposing a Program for a Multiple Processor System," International Conference on Parallel Processing, Bayne Highlands, MI, August 1980.

3. Arvind and Dennis, F.D. "New Computer Architectures," IFIP Congress 80, Tokyo, Japan, October 1981.

4. Arvind "A Dataflow Architecture with Tagged-Tokens,"

   University of Tokyo, Japan, October 1980;
   MIT Industrial Liaison Program, Diamond Hotel, Tokyo, Japan, October 1980;
   Hitachi Ltd., Kawasaki, Japan, October 1980;
   Electrotechnical Laboratories, Japan, October 1980;
   Central Research Laboratory, NEC, Japan, October 1980;
   University of Illinois, Urbana-Champaign, IL, December 1980;
   Cornell University, Ithaca, NY, December 1980;
   8th Annual Symposium on Computer Architecture, Minneapolis, MI May 1981.

5. Arvind "Inadequacy of Fortran for Parallel and Vector Machines," Fujitsu Laboratories, Kawasaki, Japan, October 1980.

6. Arvind "Managers in the Irvine Dataflow Language," IBM Homestead, Amagi, Japan, October 1980.

7. Arvind "Dataflow: An Alternative to von Neumann Languages and Architectures," 8th Annual Symposium on Computer Architecture, Minneapolis, MI, May 1981.

8. Arvind "The U-Interpreter" 8th Annual Symposium on Computer Architecture, Minneapolis, MI, May 1981.

9. Arvind "Data Structure in Functional Languages," 8th Annual Symposium on Computer Architecture, Minneapolis, MI, May 1981.

INFORMATION MECHANICS

Academic Staff

E. Fredkin, Group Leader

Research Staff

T. Toffoli

Graduate Students

R. Giansiracusa
N. Margolus
D. Payton
A. Ressler

Undergraduate Students

R. Fearing
A. Hoffman
W. Pong

Support Staff

R. Hegg
A. Schmitt

Information Mechanics

1. CONSERVATIVE LOGIC AND REVERSIBLE COMPUTING

One of the goals of conservative logic is to explore ways of realizing virtually nondissipative computation. A well-known thermodynamical argument shows that the erasure of one bit of information in a physical system entails the dissipation of an amount "kT" of energy. Since ordinary logic elements such as the NAND gate destroy approximately one bit of information at every step, it seemed that the "kT" barrier set a definite lower bound on amount of energy that a computer must dissipate.

We have shown that this argument does not apply to reversible logical networks, and we have proved that universal computing capabilities can be achieved in such networks without a substantial increase in complexity with respect to conventional logical networks. However, the first model that we used for that purpose ("conservative logic," based on the Fredkin gate) was an abstract mathematical one involving binary variables and Boolean functions. It was important to show that this model could be made more physical-like, using primitives compatible with the rules of analytical mechanics and based on stylized but recognizable physical effects.

In the past year, we have developed and extensively analyzed a model having the desired properties, namely, the billiard-ball model of computation. The primitives of this model are identical to those used a century ago to give a microscopical, statistical-mechanical account of the properties of perfect gases. Namely, the model employs identical hard balls which collide between themselves and with the walls of a hard container. In our case, the container is given an appropriate shape (which corresponds to a computer's hardware) and the balls are given appropriate initial positions and velocities (corresponding to a computation's software program and data). What is surprising is that a simple physical effect such as the collision of hard balls (which can be taken as a prototype for more realistic physical effects, such as inverse-square-law interactions) is sufficient to provide a nonlinear interaction usable for the systematic design of digital information processing.

The billiard-ball model is a classical-mechanical one. Recently, we have started working on quantum-mechanical models of conservative logic. In particular, a graduate student, Norman Margolus, is making much progress in simplifying and generalizing Benioff's approach to quantum-mechanical computation.

On one hand, such models are necessary in order to answer many questions concerning the influence of external noise, system initialization, read-out, etc. On

the other hand, only in a quantum-mechanical model can one effectively deal with another potential obstacle to nondissipative computation, namely, the alleged "hT" barrier (where h is the quantum of action). Actually, the arguments for such a barrier are rather shaky, but constructive counter-arguments require a detailed working model.

Moreover, while certain issues concerning the interpretation of quantum mechanics (e.g., the well-known "measurement" problem) can be looked on as rather pedantic questions in many experimental settings, they become vital issues in the context of microscopic computation.

A very exciting and fruitful event has been the conference on "Physics of Computation," (May 1981) sponsored by the MIT Laboratory for Computer Science and organized by our group in collaboration with Rolf Landauer of IBM Research. Some of the most conspicuous participants: Dyson, Feynman, Wheeler Landauer, Keyes, Bennett, Finkelstein, Greenberger, Benioff, Petri, Zuse, Minsky, and Tribus. The conference showed that there is a strong agreement between physicists and computer scientists as to the essential problems of physical computation, and confirmed the soundness and relevance of our research approach.

2. SEMI-INTELLIGENT CONTROL

Semi-intelligent control is an original approach to the problem of controlling machinery through the use of a distributed network of microprocessors. This approach stresses the use of local, uniform, and redundant information to drastically reduce bandwidth, and the systematic use of look-up tables (rather than analytic methods) to build up flexible learning and performance in control tasks.

During the past year we have started a number of pilot activities in this area. Using a network of 6801 one-chip microprocessors, we have experimented with two implementations of serial broadcast busses. We have done some work on obtaining high performance from inexpensive DC motors and similar actuators through the use of "personality" tables in the feedback loop.

A graduate student, Robert Giansiracusa, has continued working on physically-based learning strategies for the Hinge System; another graduate student, Willie Pong, has completed the mechanical design for the actual prototype of the Hinge System, and has started experimenting with this life-size prototype.

Publications

1. Fredkin, E. and Toffoli, T. "Conservative Logic," MIT/LCS/TM-197, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

2. Pallottino, S. and Toffoli, T. "An Efficient Algorithm for Determining the Length of the Longest Dead Path in a LIFO Branch-and-Bound Exploration Schema," to appear.

3. Toffoli, T. "Reversible Computing," Proceedings of the 5th International Symposia on Automata, Languages, and Programming, Springer-Verlag, Holland, 1980, 632-644.

4. Toffoli, T. "Physics and Computation", to appear.

Theses Completed

1. Ressler, A. "The Design of a Conservative-Logic Computer and a Graphical Editor Simulator," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1981.

MESSAGE PASSING SEMANTICS

Academic Staff

C. Hewitt, Group Leader

Research Staff

H. Lieberman

Graduate Students

G. Barber
E. Ciccarelli
W.Kornfeld
P. Koton

Undergraduate Students

B. Pines
D. Theriault

Support Staff

B. Dexter

Visitors

G. Attardi
M. Simi

Message Passing Semantics

1. INTRODUCTION

In this report we develop the semantics of workstation networks in the office in terms of the concepts of application structure and organizational structure of the office. Application structure is concerned with the rules and constraints of the domain of office work such as accounting, law, or social security regulations. Organizational structure is concerned with the social structure of the office as an organization and as such concerns the subsystem components of the office and roles of office workers. Detailed knowledge of office application structures and organizational structures is necessary in order to understand how they interact and evolve.

Problem solving is a pervasive activity within offices. It is necessary within the application domain to fulfill the requirements of the application tasks and it is necessary within the organizational domain to understand the influence of the structure of the organization on the application domain. Problem solving is also performed when office workers apply general knowledge about office procedures to the specific cases encountered in their daily work.

We discuss how a description system (named OMEGA) can aid in the construction of interactive systems whose intent is to describe the application and organization structures. Using the knowledge embedded within itself about the office, OMEGA can help support office workers in their problem solving processes.

2. WORKSTATION NETWORK SEMANTICS

Although the computer has been used in the offices for many years, its use has been limited mainly to highly structured and repetitive tasks in a non-interactive environment. Today we see the use of the computer in a wider variety of areas in the office. Word processing systems, electronic mail systems and other tools based on digital computers are finding their way into the office space proper. Office workers are beginning to have first hand experience using computers. Computers are no longer the mystical beasts available only to an esoteric few.

With this change in the tools available an office worker has come a realization that there is enormous potential in the use of the computer--especially in networks of interconnected workstations-- in novel and, as yet, unforeseen ways. These new uses will impact the way office work is done in fundamental ways; demanding new

ideas about how to manage information in an office, and a new conceptualization of what office work is in the presence of new computational capabilities.

As a step toward understanding the impact of this expansion in the use of computers in the office we propose to investigate *workstation network semantics*. In our view, workstation network semantics encompasses the study of the two dominate structures in the office--the application structure and the organizational structure--and how these structures interact. The basis of the modes of inter- and intra-structure interaction is through communication of information. Thus communications have content or *meaning* in terms of the application structure and the organizational structure. We couch this meaning or the *semantics* of the communications in terms of the effect of these communications on the subsequent behavior of the office system.

Formalizing and studying the application and organizational structures of office systems is an important goal of our research. We intend to develop a formalization which is implementable on a computer and which has well defined semantics. This has advantages from two perspectives. A formalization allows us to talk about what offices are and what offices do in a more precise manner, free of the ambiguities and imprecision of informal language. With a formalization that has computational underpinnings we can embed the knowledge expressed in the formalization within a computer system itself. Thus we are able to embed knowledge about office systems within the computational systems used in offices. Our belief is that this approach will greatly enhance the capabilities of office computational systems.

Let us consider what we mean by the application structure of an office system. The application structure concerns the subject domain of the office. It comprises the rules and objects that compose the intrinsic functions of a particular office system. In an office concerned with loans, the application structure includes such entities as loans, credit ratings and rules such as criteria for accepting or rejecting loans. The application structure of an insurance company is concerned with insurance policies, claims and actuarial tables. The application structure explains the scope of the functionality of an office system on a subject domain as well as providing a model by which those functions are characterize. The application structure is, overtly, the primary reason for the existence of the office.

In contrast to the application structure we have the social structure. Our concern with this aspect of an office system stems from the fact that the activity in the application domain of an office system is realized by people cooperating in a social system. We consider the organizational structure to include both the formal organizational structure and the informal structure of social relations between the members of the organization. The system of formal controls and lines of authority in an organization have a complementary structure of informal relations among the office workers [1].

To develop the formalism we need to describe the structures in an office system we draw on ideas and theories from the field of artificial intelligence. The formalism we are developing allows us to embed knowledge within a computational system and reason using this knowledge. This allows us to describe and reason about the application structure and the organizational structure. As we describe in the body of this paper, the use of a computational description language has many advantages. Its major benefits with relevance to our discussion here are that it will allow the use of computational systems in weakly structured, knowledge rich environments and that it provides a precise language within which to characterize office systems.

In the following pages we describe some of the important issues in our research on workstation networks. The major emphasis is on organizational structure. We feel that the reason many of the past efforts have been less than successful is due to an overemphasis on the application structure. In this paper we argue that the social structure of the organization has a direct effect on the performance of the organization and that this in turn affects the way new technologies are accepted and used. In the next section we elucidate this point. In the third section we discuss the relationship of organizations theory and artificial intelligence; and we discuss the nature of work in the office and the technology with which this work is accomplished. In the fourth section we describe OMEGA, our knowledge embedding language, and describe its use in the *knowledgeable office system*. The fifth section discusses the *Actor Model of Computation*, the underlying computational framework upon which the technical aspects of our studies are based. The last section describes how each of the subjects in the previous sections need to be combined within the Knowledgeable Office System to form an *interactive* and *integrated* environment.

3. ORGANIZATIONAL STRUCTURE

The study of organizational structure is an important aspect of office systems which has been largely neglected in past efforts to introduce computers into office systems. With electronic office systems as intimately embedded in offices as we propose, this neglect is no longer possible. Below we consider some concepts central to this view of office systems.

3.1. Relation to the Environment

An important characteristic of organizations is that they exist in an environment, constantly interacting with and dependent on it. The behavior of an office depends not only on its input conditions but on the conditions that exist in the extra-office environment. In an accounting office the formally required output may be audits, but how these audits are created and what they mean depend on tax laws, legislation

concerning accounting procedures and the currently accepted body of knowledge about accounting practices. A report of a business entity's financial status has meaning with respect to the process that was used to create it as well as the processes that are used to interpret its significance.

3.2. The Tangibility of the Electronic Office

Computer based office systems are moving in the direction where all office work will be done on or through the computer system. This has profound implications on the way information in the office can be manipulated, stored and accessed.

Mass storage technology is such that large quantities of data can be inexpensively stored compared to paper-based storage methods such as file cabinets. This simply means that, for the same price, the volume of information that can be kept is larger. This trend will continue in the future.

An important difference between paper- and computer-based storage technologies is accessibility of information. In the computer-based system not only can more information be stored for a decreasing price but it can be accessed more quickly and more flexibly than in the paper-based system. This affects the way work in the application structure of the office can proceed, but it also affects what the office can know about its own performance. Detailed historical records can be kept and referenced. This adds a dimension of tangibility to the office not present in paper-based systems. Performance of the office can be monitored and used to control office activity. However, as we discuss in the next section, this can cause problems as well as benefits.

3.3. Measurement of Performance

As we saw in the previous section, electronic office systems can keep detailed databases of information on the performance of their organization and the individuals within it. This information is useful for *regulatory functions* which gear office work to certain factors such as production demand. Performance information is also useful for the *adaptive* purposes which seek to help the office evolve so that it may continue to survive in a changing environment. However, care must be exercised about what information is kept and how it is interpreted.

Numbers are exceedingly easy to collect in an electronic office system, but if these numbers are used to drive an adaptive or regulatory mechanism, it is essential that an attempt be made to analyze the effect on the future behavior of the office. If this is not the case the resultant behavior may not reflect goals of the organization.

A major problem here is that there is little understanding about how offices work in their day to day operation. Initial performance measurement often points out surprising discrepancies between the believed and actual office performance characteristics. The temptation to enforce a particular behavior on an office must be resisted until the implications of the change are well understood. This is particularly true with regard to the effects of an enforced behavior on the social structure of an office.

3.4. Conflicting and Common Interests

Another important function within an office system is the making of decisions (which we will call the *authority structure* of the office system). A common authority mechanism within offices is a system of checks and balances or *controls* between offices charged with advancing somewhat conflicting interests. An important strategy for maintaining balance is to establish separate groups in an adversarial relationship within an organization to look after conflicting interests. Policies are then established and evolved by negotiation. This strategy is often used in preference to the alternative of attempting to have one group attempt to "rationally" balance the conflicting interests.

Accounting systems are an example where controls are maintained by adversarial relationships between different groups. In many cases accounting systems are required to have certain controls by law, for example. As a result some proposed computerized accounting systems would be illegal to use. This requirement influences the design of office systems by placing a constraint on information flow and requires that office systems be designed so users cannot violate these information flow constraints [2].

Systems of common interest are used to advantage in offices. It has been noted [1] that workers cooperate better and form stronger social relationships if they share the goals of a task and are mutually dependent on each other to achieve the goals. Care must be taken to avoid inadvertently upsetting these systems of controls and dependencies. .

3.5. Dangers of Separation

Let us consider the situation where word processing centers were introduced in an attempt to increase productivity of typed documents.

The traditional view of secretarial tasks is pictured as comprising such tasks as answering telephones, taking messages, performing administrative duties, making appointments, and typing documents. Word processing equipment is introduced

with the intent that operators be trained in the use of word processing machines and be charged with typing whatever documents are delivered to them. The rationale behind this approach was that operators would become proficient at document production with the aid of word processing machines and that secretaries would not have to be concerned with document production, freeing them to perform their other tasks more efficiently. The hope was that in this way the overall productivity of the office would increase.

To the surprise of some, it has been found that the introduction of word processing centers into an organization often has an adverse affect on the production and quality of work in the organization. This stimulated interest in introduction strategies to more carefully control adverse effects. The introduction of word processing centers has had the effect of separating individuals from the semantics of their tasks. The text typed often has almost no meaning beyond the word level to the operators so it is impossible for them to detect important errors and ambiguities and resolve them. The operators have little knowledge about the tasks they are performing; they cannot be as knowledgeable and involved in the task as a secretary who has personal knowledge of the semantics of the material to be typed.

This problem can be explained in terms of a more careful inspection of a secretary's tasks. The secretary's tasks, as expressed by the expectations of his or her coworkers, not only involve those tasks mentioned above but include verification and correction of the information the secretary is concerned with. This stems from the fact that information is often incomplete, ambiguous or in error. The secretaries are familiar with the semantics of the information with which they are working. They know acceptable levels (via norms) of error, ambiguity and incompleteness.

A more subtle problem that arises from the separation of the word processing centers is that they become entities which interact with their customers in more formalized and less flexible ways. The social fabric of the organization changes in such a way as to introduce new authority and managerial issues. This has political implications when information that is likely to be misinterpreted flows outside of the sphere in which it is understood.

3.6. Effects of Social Structure on Organizational Performance

The performance of an organization is directly influenced by the informal social structures among its members. For example:

- The decisions an individual makes that affect a coworker are based in part on the social relationships between the workers. They include the individual's trust in the coworker, his assessment of the coworker's

competence, his beliefs about what the coworker knows and his knowledge of the coworker's habits.

- When individuals depend on each other to accomplish the same goals, the informal working relations are strongest and the common goal is most easily accomplished. In the case where the relationship is less bidirectional, establishment of the goal becomes a more difficult task to the point that formal sanctions may be necessary to insure that the goal is accomplished properly and in a timely manner.

- Pools of office workers, where each worker is performing the same task, tend to form their own informal social hierarchies. The more experienced and skilled workers tend to be accepted as the informal leaders and representatives of the groups. These informal leaders are the ones most likely to form working relationships with managers of the work pools. Via these relationships, decisions are made and strategies are planned.

When a new piece of machinery is introduced, workers must learn about the technical aspects of the machinery as well as new dependencies and informal understandings. Workers generally learn this kind of information from more experienced members of the office. In the case of new machinery there may be no experienced members and a learning period in which the dependencies and understandings are evolved must be entered. Thus the introduction of new technology will affect both application structure of the office and the informal social structure. The neglect of the social impact of new technology has caused many problems in the introduction of systems into the office in the past.

4. THE NATURE OF OFFICE WORK FROM AN AI PERSPECTIVE

Of concern to us here is the behavior that organizations exhibit. Organizational behavior is often behavior that is considered intelligent in humans and includes such activity as problem solving, knowledge acquisition and manipulation, and adapting to a changing environment. Organizations exhibit behavior that can neither be implemented given current AI programming methodologies nor can it be explained by current AI theories.

There are many reasons why the study of organizational systems are of interest to AI researchers. Organizations are accessible in a way that humans are not. It is possible to examine the workings of an organization in more detail than it is possible to examine the processes by which a human solves a problem or understands natural language. An organization can be metered, analyzed and experimented with

in ways that are not possible with humans. Hypothetical organizational structures can be implemented and examined.

There is a continuum of scale when considering organizations that is not present with humans. At one end of the scale we have an organization composed of a single human. At the other end are organizations composed of many thousands of individuals. This continuity is interesting from at least two points of view. First, we may see how functions present in individuals can be implemented using groups of individuals when the complexity or scope of the functions exceeds the capacity of a single individual. Second, we see various ways in which the functions that organizations perform can be factored as the size of the organization increases.

Many issues that arise in computer science and artificial intelligence also arise in organizations theory. These include distribution vs. centralization of resources; coordination and synchronization between processes; control systems; information flow; abstraction and controlling complexity; adapting to a changing environment; knowledge use, manipulation and representation.

The study of organizational systems is relevant to the current interest in the communicating experts metaphors in AI research [3]. In these metaphors it is assumed that the complexity and sophistication of human intelligence arises out of interactions between simple entities or entities of a limited domain of expertise. This is a metaphor readily adaptable to the study of organizations.

4.1. The Pervasive Nature of Problem Solving

Problem solving is a pervasive aspect of office procedures which has been neglected until very recently [4] [5]. Understanding this problem solving activity is a prerequisite to developing systems which aid in performing tasks that previously have not been amenable to computer processing. Several situations give rise to problem solving activity on the part of office workers. Problem solving is often required within the application domain. Decisions are made concerning the best way, according to some criteria, of obtaining a result. A common task requiring problem solving is to try and diagnose abnormal results of an office procedure. In this case it is necessary to reason about the progress of a procedure in an effort to pinpoint the cause for the anomalous behavior. Once this is done, further reasoning is necessary to determine what the abnormal effects of the procedure were and how to compensate for them.

Problem solving also arises from the fact that the office exists in an environment and constantly interacts with that environment in implicit as well as explicit ways. Changes in the environment must be detected and compensated for. An accounting office's avowed functionality has little to do with a paper forms supplier or the postal

service. But accounting offices frequently interact with these organizations and if these organizations do not behave normally, compensatory action must take place in the accounting office.

This conception of office activity differs from the traditional view that office activity consists of a sequence of well defined steps. Indeed, some office activity does have this characteristic. The areas where computers have made a significant impact, such as accounting and inventory control, are areas that are highly structured and repetitive, thus easily formalized in terms of a sequential model. By considering the office from a problem solving perspective, we relax the rigid requirements on tasks performed by computers. Important aspects of this view of office activity are:

- **different sets of goals** that evolve over time (these are often implicit in the office procedures and often ill defined);

- **problem solving mechanisms** by which goals may be satisfied in their proper order at the appropriate time;

- **constraints**, derived from the organizational and application domains, within which the office procedures must work.

A difficulty in formally defining the content of office work exists because office workers use their ability to plan and execute, in the face of unexpected contingencies, actions that achieve the goals of office work. What is really desired is the knowledge that drives the planning process and knowledge about how the problem solving process works.

More knowledgeable office systems can help the office workers by supporting them in their problem solving activity. Analysis of past activity helps diagnose abnormal office procedures and descriptions of postulated activity help determine the consequences of future actions. With descriptions of tasks embedded within a computer system, the computer system can aid the office worker. The computer system can determine what the goal of current activity is, what possible ways may exist for achieving the goal and when the goal is actually realized.

4.2. Explicit Representation of Goals and Constraints

Office workers are able to handle unexpected contingencies in their daily work because they know the goals of the office work and because they know the constraints that must be maintained during the execution of the office work. These goals and constraints are often implicit in the work and in the office workers' knowledge of their work. Thus it is hard for a computer or another human being to

understand the decisions an office worker makes in planning a problem solving strategy to handle unexpected contingencies.

To support the problem solving activity in office work, knowledge about the goals and constraints of the office work is explicitly represented. This builds a teleological structure of the office work within the computer. Actions that would be performed during the course of the office work are linked to the reasons they are performed and to the constraints that they are required to maintain. Explicit representation of the goals and constraints exposes hidden assumptions about the office work and makes the actions performed by an office worker more understandable by machine or by another individual.

The explicit representation of goals and constraints provides a recourse to handle unexpected contingencies. If a particular action cannot be performed, the computer system can possibly suggest an alternative action. Failing this, the office worker can use the computer system to examine the goals and constraints of an action that cannot be performed. Together, the office worker and computer system can construct a new plan of action that maintains the necessary constraints and makes progress toward achieving the goals in question.

4.3. Organizations Theory and AI

Our underlying interest in the study of organizations is to consider the relationship between the technology used to accomplish work in the office and the work that needs to be done. The characteristics that technology for the office must have can be derived from several considerations. First, using people vs. using people and machines to accomplish the knowledge processing; second, the open-ended character of knowledge in the office world; and third, the resource consuming nature of decision making in order to achieve goals.

One can ask the questions "What have the years of study in organizations theory produced?" "What can artificial intelligence contribute?" and "Is the wheel about to be reinvented again?" To answer these is questions we consider the following view of organizations. There is a kind of work that organizations--especially information intensive organizations such as offices--perform, and a technology by which this work is accomplished. By and large, the technology by which the work is accomplished has largely consisted of paper-based and verbal communication, paper-based storage of information, and the members of the organization. The relationship between the work that offices accomplish and the technology used to accomplish it has not been of concern because it has not changed until recently. Thus organizations theory has not dealt with the question of the relationship between work in the office and how it is done. Much can be gained by examining the work in the office as knowledge manipulation and problem solving activity.

The relationship between work and work technology has been an issue in more routinized, production line style, non-information related tasks. As a result there has been much study in the name of management science and industrial engineering. Within the office there has been the use of centralized computer facilities for accounting and inventory. Both of these functions have a highly structured and rigid interface to the workers in the office. In their capabilities they are extensions of the paper-based systems. Technology impacting the work in the office has been limited to devices such as the batch computer facilities, telephone, typewriters and recently, word processing. The introduction of each of these has impacted the way office work is done. The impacts have been handled on a case by case basis; no theory exists of what happens when new technology is introduced. The unpredictable results of the efforts to introduce word processors into offices is testament to the fact that both the relationship between technology and office work is not well understood and that office work itself is not understood. In the cases of the technologies mentioned above, the work in the office--the thinking, the knowledge processing--has not been impacted in any significant way. Certainly not as drastically as it will be in the years to come.

5. THEORETICAL FOUNDATIONS

In this section we discuss the theoretical foundations of workstation network semantics. We first consider the description system OMEGA, the knowledge embedding language. Following this, we discuss the concurrent systems theory that forms our foundation for understanding and building distributed computer systems.

5.1. The Description System OMEGA

We are developing a description system (OMEGA) to embed knowledge about offices into an electronic office system [6] Descriptions are used to describe the properties of objects in an office. Within an office system, descriptions are used to embed knowledge about office procedures and the tasks of office workers, as well as replace current day paper forms. Descriptions perform several functions that were heretofore entrusted to forms such as:

- **Storage of information** as in records.
- **Transfer of information** as in messages.
- **Display of information** in an abstracted and structured manner.
- **Accumulation and modification of information** as the form is used by individuals in the accomplishment of their tasks.

Descriptions provide some of the functionality of an automated forms flow system. Descriptions are a very general facility; one of their uses is to support electronic forms but they are used for much more general knowledge embedding purposes.

Descriptions are of underlying importance within OMEGA; they express relationships between the objects in the electronic office system. A form is the visual manifestation of a description. An electronic system with descriptions stores the information contained in descriptions in an inheritance hierarchy. Those descriptions which are forms are displayed on video devices for perusal and modification. In addition to the capabilities supplied by forms, descriptions function in additional capacities:

- Descriptions are a **means for error checking** of information in an office system.

- Descriptions are a **basis for retrieval** of stored information.

- Descriptions are a means by which the **structure of the application and organizational domains** of an office system are specified.

- Descriptions determine the **semantics** of entities in an office system via their specified relationships to each other.

- Descriptions relativized to **viewpoints** are a means of dealing with change and avoiding inconsistent states.

The added dimension descriptions give to an office worker is exhibited in the following example. An office we have studied which is part of the Department of Defense is one in which officers are assigned to new tours of duty after their current assignment expires. Often in this system an assignment officer is asked questions about data in forms such as: "How many officers above the rank of captain are at sea and are due to roll within the next six months?" Questions of this type have the characteristics that their specifics cannot be anticipated and that they require a tedious, time-consuming search of large amounts of data. A retrieval facility allows a user to fill in an example description with variables and conditions and use the example description to match against stored descriptions. This scheme gives a user the power to easily express a wide variety of questions similar to the one above. It is related to but more general than such systems as Query By Example [7] in that information exists in a semantic hierarchy and thus may be accessed in terms of its semantic properties as well as in terms of predicates on the information itself.

A mechanism supplied by OMEGA is the viewpoint mechanism. Viewpoints are a means by which to relativize descriptions to time. Thus they are used to indicate

when a description is applicable. Viewpoints themselves are descriptions and thus there is full generality in describing viewpoints and the relationships between viewpoints.

Descriptions provide a means by which to embed knowledge about offices and office procedures within an office system. We refer to such a system as a *knowledgeable office system*. The structure of office procedures is described in terms of their goals, the environmental constraints under which they must operate and the tasks of individuals involved in those office procedures. This knowledge can be used in many ways. It can be used to predict what information may be needed by the office worker as he attempts to solve the problems posed to him by his tasks. Descriptions form a basis within which to express and maintain the status of goals and the relationships between interacting goals. In an interactive environment, descriptions serve as a basis within which to interpret basic commands and commands programmed by the user.

The office worker must be able to program his workstation to help him accomplish his tasks, but this programming must be done in a different manner than it is currently. It is undesirable that someone concerned with assigning officers to new duties communicate with his workstation in terms of integer variables or iteration constructs. The worker must be able to communicate in the language in which he thinks and he must be able to develop programs in as painless a fashion as possible. An alternative to the traditional programming practices is a methodology known as *concrete programming* [8] In this approach, a user defines the effects of a program in a piecemeal fashion by using operations on concrete example data items in a manner similar to the way he would normally perform the procedure. This allows the user to see the effects of his program as he builds it, partially dissolving the dichotomy between running and writing programs. In this manner, programmed office procedures emerge from solving concrete problems in the course of daily work.

5.2. Concurrent Systems

As a computational framework for our ideas, we are developing the Actor theory of computation. Part of this work involves the design of programming languages like ACT1 [9] and ETHER [10] and partially involves the mathematical definition of the semantics of these programming languages. In two areas of concern we feel that a language with well understood semantics is necessary for the design of office information systems; these pertain to *guarantee of service* properties and the implications of the order of arrival of messages.

Whenever communicating programs execute on a computer system the problem

of guarantee of service arises. Guarantee of service is important to insure that in situations where requests are constantly competing for a system's resources, all requests made are serviced. Thus, within the office environment consider a case where many loan applications are submitted to the office system over a period of time. A property one would desire to prove is that each loan application submitted will be processed and in time will result in a response, be it an acceptance or a reason for rejection. It is a theoretical property of some computational models that guarantee of service cannot be insured. An advantage of building a system in the Actor model of computation is that guarantee of service properties can be established and implemented. For example, in [9] an implementation of a hard copy server is given along with a proof of guarantee of service.

An additional reason for the importance in providing a precise mathematical definition of a programming language to be used in an office system is that the meaning of the different kinds of messages arriving at workstations and the actions they evoke are very dependent on the order in which the messages arrive. Concurrent systems theory supplies the concepts with which to talk about the arrival orderings of messages and the consequences of the possible arrival orderings.

Actor theory formalizes and describes the behavior of objects called actors as they communicate via message passing. In this model, all computations are represented by message passing between actors. The receipt of a message by an actor may trigger additional messages sent to other actors, thus continuing the computation. This model is particularly well suited for application to the office environment because activity in both the Actor model and the office is driven by the receipt of messages. Activity is initiated when a message is received, be it a loan application, a message triggered by the time of day, or a message that asks for the square root of a number.

The communication in the Actor model and much of the communication in offices is unsynchronized communication. The intended recipient need not be ready to accept a message before it can be sent. In an office, many messages are sent without requiring that the intended recipient be in a particular state at the time of transmission. A mail system is an example of unsynchronized communication while a telephone exchange between caller and answerer is synchronized communication.

Note that an important task a secretary performs is to answer a telephone and take messages. These messages will then be delivered to the intended recipient at a later, more convenient time. The telephone is a fast way to send messages but it requires that someone be present to answer it; it is synchronized communication. Synchronized communication places heavy constraints on the communication mechanism since both parties must synchronize before a message can be exchanged. The secretary often functions to desynchronize messages that need to be transferred quickly.

6. AN EVOLUTIONARY, INTERACTIVE ENVIRONMENT

An area where much effort is expended in an office system is in attempts to deal with change, both within the system itself and between the system and the environment it exists in. Viewpoints are a technology which we are developing to address this problem. They allow changes to be considered in a consistent manner by relativizing the information before and after the change to different viewpoints and describing the relationship between the viewpoints.

Office systems must be flexible and able to adapt to change. As workers become more adapted to the use of more sophisticated electronic office tools, deeper organizational changes may begin. As our understanding of the office increases, more applications will arise. Technological advances engender changes in hardware and software. An office system must be able to incorporate new technology as it appears. The office exists in a changing environment and it must be able to adapt in order to continue achieving its goals. For example, if the tax laws are changed it must be possible to reflect this change quickly and easily in an office system that is concerned with taxes.

An interactive, knowledgeable system has the goal of supporting the problem solving activity which takes place in offices. This requires that the system have detailed knowledge of the application structure and organizational structure of the office work. Much of this knowledge concerns the goals of individual office procedures and the constraints within which they operate.

Many facilities such as mail systems, text editing systems, and database systems are beginning to appear in the office. These products have been implemented as separate systems on timesharing computers or sometimes on separate machines. The approach of using independent systems has the limitation that shared objects are limited to character strings that are transferred via pipes or files. The result is that use of these facilities in a cooperative manner to accomplish tasks is cumbersome. If a system is going to manage office procedures knowledgeably, facilities that are used during the execution of the procedures must be in a more intimate relationship with each other.

The fragmentary nature of a nonintegrated computer system implies more than the technical problems of sharing objects between systems. Separate systems pay the penalty of contributing to incoherent and redundant systems. Often different sets of commands must be learned that have similar results or worse yet, similar commands having different effects. This results in complicated and difficult to understand systems.

Added coherence between different functional elements of a system has the

benefit that the user's actions and the goals of the office procedure can be understood in terms of each other. It is useful for the system to understand the goals in order to interpret the user's requests and suggest problem solving tools for achieving the goals. In turn the user's actions suggest what the current goals are and narrows the variety of problem solving methods and the size of the solution space.

7. CONCLUSIONS

We believe that the time has come to begin the development of workstation network semantics as a field of endeavor which studies the meaning of messages sent in an office. These messages have meaning from several points of view. These messages reflect the application structure and organizational structure of offices, including office organization, and office procedures, as well as issues of power and control that arise in negotiations. A message has a social content. A message has application content. For example, messages concerning purchase orders or requisitions must obey certain rules and regulations. From the point of view of both applications and interpersonal relations, a message has timing content. For example, a request to withdraw money from a checking account can have different consequences depending on whether it arrives before or after a deposit message.

Much of the work performed by office workers has important problem solving aspects. Future electronic office systems must support this problem solving activity. This is one reason why it has been so difficult to extend sequential, algorithmically oriented programming languages such as COBOL and PL/1 to new office applications. The goals of office procedures need to be understood by any electronic office system used by the workers. Research should be directed toward the goal of developing interactive support systems to aid office workers in their daily problem solving activities. Such systems must have knowledge of the goals and constraints of office procedures in order to provide effective support for office workers in using their workstations.

It is very important to consider the sociological impact of electronic office systems. Knowledgeable office systems must be designed to *meet* the organizational structure at the time of their introduction and then *evolve with* the organization. The negotiation activity necessary to balance interests among competing groups must be maintained. New ways of structuring the office must be judged in light of their impact on the semantics of work including the application, timing, and organizational content of messages. New ways of measuring performance need to be evaluated in terms of their impact on the semantics of the work performed.

References

1. Browner, Chibnik, Crawley, Newman and Sonafrank "Report on a Summer Research Project: A Behavioral View of Office Work," Xerox PARC Technical Report, Xerox Palo Alto Research Center, Palo Alto, CA, 1979.

2. Bailey, A.D., Gerlach, J., McAfee, R.P. and Whinston, A.B. "Internal Accounting Controls in the Office of the Future," Computer 14, 5 (May 1981).

3. Kornfeld, W.A. and Hewitt, C. "The Scientific Community Metaphor," IEEE Transactions on Systems, Man, and Cybernetics SMC-11, 1 (January 1981).

4. Wynn, E. "Office Conversation as an Information Medium," Ph.D. dissertation, Department of Anthropology, University of California, Berkeley, CA, 1979.

5. Suchman, L. "Office Procedures as Practical Action: A Case Study," Xerox Palo Alto Research Center Technical Report, Palo Alto, CA, September 1979.

6. Hewitt, C., Attardi, G. and Simi, M. "Knowledge Embedding with a Description System," MIT Artificial Intelligence Laboratory Memo, MIT Artificial Intelligence Laboratory, Cambridge, MA, December 1979.

7. de Jong, S.P. and Zloof, M.M. "The System for Business Automation (SBA): Programming Language," Communications ACM 20, 6 (June 1977), 385-396.

8. Lieberman, H. and Hewitt, C. "A session with TINKER: Interleaving Program Testing with Program Design, MIT Artificial Intelligence Memo 577, MIT Artificial Intelligence Laboratory, Cambridge, MA, April 1980.

9. Hewitt, C., Attardi, G. and Lieberman, H. "Specifying and Proving Properties of Guardians for Distributed Systems," Proceedings of the Conference on Semantics of Concurrent Computation, Evian, France, July 1979.

10. Kornfeld, W.A. "Using Parallel Processing for Problem Solving," International Joint Conference on Artificial Intelligence-79, Tokyo, Japan, August 1979.

Publications

1. Attardi, G. "Trends in the Evolution of Personal Computers," Atti del Congresso AICA '80, Bologna, Italy, October 1980.

2. Attardi, G., Martelli, A. and Montanari, U. "Il Meccanismo dei Moduli nel Linguaggio de C-net," Progetto Finalizzato Informatica del Consiglio Nazionale delle Ricerche, No.6, ETS, Pisa, Italy, October 1980.

3. Attardi, G. and Simi, M. "Consistency and Completeness of a Logic for Knowledge Representation," Proceedings of Seventh IJCAI -81, Vancouver, B.C., August 1981.

4. Barber, G.R. , "Reasoning about Change in Knowledgeable Office Systems," First Annual National Conference on Artificial Intelligence, AAAI, August 1980, 199-201.

   MIT/LCS/TM-195, MIT Laboratory for Computer Science, Cambridge, MA, 1981;
   MIT Artificial Intelligence Laboratory Memo 620, Cambridge, MA, February 1981;
   SIGART Newsletter, January 1981;
   SIGOA Newsletter, Spring 1981.

5. Barber, G.R. and Hewitt, C.E. "Research in Workstation Network Semantics," IEEE Transactions on Software Engineering, Fall 1981.

6. Barber, G.R. "Reasoning about Change in Knowledgeable Office Systems," Artificial Intelligence Laboratory Working Paper, MIT Artificial Intelligence Laboratory, Cambridge, MA, May 1981.

7. Barber, G.R. and Hewitt, C.E. "Research in Office Semantics," Introductory Readings in Expert Systems, Donald Michie (Ed.), Gordon and Breach, London, England, Fall 1981.

8. Hewitt, C.E., Attardi, G. and Simi, M. "Knowledge Embedding in the Description System, Omega," Proceedings of AAAI Conference, Stanford, CA, August 1980, 157-164.

9. Hewitt, C.E. "The Apiary Network Architecture for Knowledgeable Systems," Proceedings of Lisp Conference, Stanford, CA, August 1980, 107-118.

10. Hewitt, C. "Apiary Multiprocessor Architecture Knowledge System," Proceedings of the Joint Stanford Research Center/University of Newcastle upon Tyne Workshop on VLSI, Machine Architecture, and Very High Level Languages, P.C. Treleaven (Ed.), University of Newcastle upon Tyne Computing Laboratory, Newcastle-upon-Tyne, TR-156, October 1980, 67-69.

11. Hewitt, C.E. "Evolutionary Programming," Freeman and Lewis (Eds.), Academic Press, 1980, 133-148.

12. Kornfeld, W.A. "A Synthesis of Language Ideas for AI Control Structures," AI Working Paper 201, MIT Artificial Intelligence Laboratory, Cambridge, MA, July 1980.

13. Kornfeld, W.A. "Lisp and Music," Computer Music Journal 4, 2 (Summer 1980).

14. Kornfeld, W.A. and Hewitt, C.E. "The Scientific Community Metaphor," IEEE Transactions on Systems, Man, and Cybernetics SMC-11, 1 (January 1981).

15. Kornfeld, W.A. "Using Parallelism to Implement a Heuristic Search," AI Memo 627, MIT, Artificial Intelligence Laboratory, Cambridge, MA, March 1981.

16. Kornfeld, W.A. "Combinatorially Implosive Algorithms," to appear.

17. Lieberman, H. and Hewitt, C.E. "A Session with TINKER: Interleaving Program Testing with Program Design," Proceedings of LISP Conference, Stanford, CA, August 1980, 80-99.

18. Lieberman, H. and Hewitt, C. "A Real Time Garbage Collector That Can Recover Temporary Storage Quickly," MIT/LCS/TM-184, MIT Laboratory for Computer Science, Cambridge, MA, October 1980.

Theses Completed

1. Layson, S. "An Apiary Worker: Processor Design," S.B. thesis, MIT Department of Humanities, Cambridge, MA, June 1980.

2. Theriault, D.G. "A Primer for the Act-1 Language," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

Theses In Progress

1. Barber, G. "Supporting Problem Solving in a Knowledgeable Office System," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1981.

2. Ciccarelli, E. "Presentation Based User Interfaces," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1982.

3. Pines, B. "Picturesque: A Graphics Editor for the Lisp Machine," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

Talks

1. Attardi, G. "Consistency and Completeness of a Logic for Knowledge Representation," Istituto de Elaborazione dell'Informazione, Pisa, Italy, March 1981.

2. Attardi, G. "The Case of Logic in Knowledge Representation," MIT Artificial Intelligence Laboratory, Cambridge, MA, April 1981.

3. Barber, G. "Reasoning About Change in the Office," 19th Annual Workshop on Office Systems, Lake Arrowhead, CA, September 1980.

4. Barber, G. "Office Systems Design and Future Technologies and Their Impact on the Office Environment," Office-80 Workshop on Office Automation, University of Toronto, Canada, September 1980.

5. Hewitt, C. "A Highly Parallel Architecture for Actor Systems," Workshop on Applicative and Parallel Computation, MIT Endicott House, Dedham, MA, July 1980.

6. Hewitt, C. "The Apiary Network Architecture for Knowledgeable Systems,"

   MIT VLSI Seminar, Cambridge, MA, September 1980;
   First LISP Conference, Stanford University, Stanford, CA, August 1980.

7. Hewitt, C. "Apiary Multiprocessor Architecture Knowledge System," Joint Stanford Research Center/University of Newcastle upon Tyne

Workshop on VLSI, Machine Architecture, and Very High Level Languages, University of Newcastle upon Tyne Computing Laboratory, UK, October 1980.

8. Hewitt, C. "Pitfalls of Office Automation." Office Automation Conference, Houston, TX, March 1981.

9. Hewitt, C. "Is Information Cost Free?" Office Automation Conference, Houston, TX, March 1981.

10. Kornfeld, W. "A Scientific Community Metaphor," Distributed Problem Solving Workshop, MIT Endicott House, Dedham, MA, July 1980.

11. Kornfeld, W. "The ETHER Model of Computation," Workshop on Applicable Languages, MIT Endicott House, Dedham, MA, July 1980.

12. Kornfeld, W. "Techniques of Parallel Problem Solving,"

    Stanford Research Institute, Menlo Park, CA, August 1980;
    Texas Instruments, Dallas, TX, August 1980.

13. Kornfeld, W. "Symbolic Music Editing," International Computer Music Conference, Paris, France, February 1981.

14. Lieberman, H. "An Example for Example-Based Programming,"

    Xerox Palo Alto Research Center, Palo Alto, CA, July 1980;
    Istituto di Scienze dell' Informazione, University of Pisa, Pisa, Italy, October 1980.

15. Lieberman, H. "A session with Tinker: Interleaving Program Testing with Program Design,"

    1st LISP Conference, Stanford University, Stanford, CA, August 1980;
    Institut fuer Informatik, University of Stuttgart, West Germany, November 1980;
    Institutonen for ADB, University of Stockholm, Sweden, November 1980;
    Uppsala Programming Methodology and Artificial Intelligence Lab, University of Uppsala, Sweden, November 1980.

16. Lieberman, H. "A Preview of Act-1,"

    Uppsala Programming Methodology and Artificial Intelligence

Lab, University of Uppsala, Sweden, November 1980;
Institutonen for ADB, University of Stockholm, Sweden, November 1980;
Software Systems Research Center, University of Linkoping, Sweden, November 1980.

17. Lieberman, H. "A Real Time Garbage Collector That Can Recover Temporary Storage Quickly," Institutonen for ADB, University of Stockholm, Sweden, November 1980.

18. Pines, B. "A Graphics Editor for the Lisp Machine," MIT Artificial Intelligence Laboratory, Cambridge, MA, May 1981.

OFFICE AUTOMATION

Academic Staff

M. Hammer, Group Leader

I. Greif

Research Staff

T. Anderson
R. Ilson

M. Sirbu

Graduate Students

J. Bakopoulos
B. Berkowitz
E. Gilbert
M. Good
J. Kunin
B. Niamir

L. Rosenstein
S. Sarin
S. Schoichet
J. Sutherland
S. Zdonik

Undergraduate Students

L. Alperin
E. Munro
J. Nitchman
M. Pelkie

V. Richman
A. Tallian
F. Tino
C. Zarmer

Support Staff

E. Balayan

M. Swanson

Office Automation

1. INTRODUCTION

Our research this year has addressed three different aspects of the broad area known as "office automation": the design of an office workstation, emphasizing multi-functionality, integration, and ease of use; the development of techniques to support the construction of office-specific, *functional* automation systems, which focus directly on the business mission of multi-person information work, leading towards the design of systems that support cooperative work in an automated environment. Though ostensibly unrelated, our work in each of these areas shares a common set of philosophical premises and emphasizes a common set of concerns.

2. AN INTEGRATED OFFICE WORKSTATION

One of the major goals of our research program is the development of a multifunctional office workstation that provides a wide range of powerful capabilities and that provides a consistent and simple interface to its users. This workstation is to be relatively hardware independent; our objective is to implement it on a number of platforms.

Our prior efforts in this area focused on the design of Etude, a novel word processing facility that provides functionality approaching that of a phototypesetting system while presenting the user with a simple interface and a supportive working environment. This year, we have built on this earlier work to begin the development of a multifunctional system, of which a revised Etude will be one component. Other components will include a database management system, and a graphics and image handling facility. These subsystems will all share a common set of basic modules for user interface handling, screen management, file handing, and the like. The architecture will contribute to the integration of these components into a coherent system.

We have also begun work on the extension and evaluation of Etude, providing it with a more general pagination and layout capability, assessing its ease of use, and laying the groundwork that will enable us to combine text with other forms of information.

2.1. Evaluating Etude

One of the major goals of Etude was that it should be easy to use. However, the "easy to use" claim is one that is made for almost all contemporary systems. How was ease of use taken into account in the Etude system design? What criteria are we using in our claim that Etude is easy to use? How can we obtain empirical evidence measuring Etude against these criteria? Michael Good has addressed these questions over the past year.

The original Etude prototype was designed by surveying a number of text processing systems and adapting their best features to meet the goal of producing an easy to use yet highly functional system. In developing the second version of Etude, another approach has been taken. A large set of guidelines for building easy to use systems was assembled, and the initial version of Etude was measured against these guidelines. In most cases, our system met these criteria. In those cases where this evaluation indicated deficiencies, some changes in syntax and vocabulary were made to improve the user interface.

Putting together a collection of ease of use guidelines is not an easy task. There are few good sources of useful principles and even fewer that have been substantiated by experimental evidence. Furthermore, the literature that does exist is very fragmented. This situation has improved in the last year or two, with some experimental reports finding their ways into widely read journals. But the general nature of knowledge in the area of user interface design more resembles an accumulation of folklore than it does a body of well-organized engineering principles. One should be aware of the knowledge that exists but avoid putting undue faith in it.

When measured against available knowledge, most parts of the original version of Etude appeared quite good. This information was principally gleaned from the human factors literature, especially that concerned with on-line computer system design. One experiment showed that a text editing system with an English-like syntax (based on a set of English verbs combined with various objects) is both easier to learn and more efficient to use than an identical system with a more traditional computer syntax. Many researchers recommend verb/object syntax and argue for the inclusion of help, menu, and undo facilities and the provision of confirmation; Etude complies with these principles. Etude was also carefully designed and constructed to provide the user with rapid feedback after each and every keystroke, even when an operation might cause a delay in the display; this is a feature recommended by several sources. Other features, such as command line editing and the listing of the last few completed operations in response to a help request, have been proposed by various authorities.

One area in which the Etude prototype did not fare so well was in the choice of names for several commands. Many authors warn of the need to choose names that do not convey adverse, obscure, or incorrect connotations. Yet Etude asked the user to **execute** commands, a term with unpleasant associations. It also allowed the user to **merge** regions of text together, though the word "merge" is seldom found other than on highway signs. Remnants of computerese cropped up in parts of the user interface (e.g., **delete** and "hlto"). As a result of this analysis, **execute** has been renamed **go ahead**; **merge** has been replaced by **combine**; **delete** has given way to **erase**; and the now obsolete acronym "hlto" has been succeeded by the word "component."

Other aspects of the user interface required more substantial changes. The **help** and **menu** facilities were made more useful by treating them in a unified manner. This was done by providing a *query-in-depth* facility, which enables a user to get progressively more detailed information about a topic or situation by choosing a menu item and pressing the **help** key. Another source of concern was the bewildering array of formatting commands. Replacing these special purpose commands with a set of more general verbs that can be combined with objects makes the interface smaller and clearer, as well as providing a more solid basis for integration of additional subsystems with Etude.

In the near future, we will be evaluating Etude to see if it is in fact easy to use. Since nearly all systems claim to be "easy to use," one must ask what is meant by this phrase. Four questions serve as examples of its connotation:

1) Can novices learn the system quickly?

2) Can it be used efficiently once it has been learned?

3) Does it make the user feel "at ease?"

4) Do people enjoy using the system?

These questions are all relevant to Etude's concept of ease of use, especially to the goal of providing a system that is easy for the novice to learn while not encumbering the expert. Question 1 reflects the goal of ease of learning while question 2 reflects ease of use. Question 3 deals with the "anxiety factor," associated with the system, while question 4 deals with users' attitudes towards it. In many systems, ease of learning and ease of use are antithetical. For Etude to meet its ease of use objectives, we must show that this need not always be the case.

Before designing the actual evaluation process, these general questions have had to be refined into more specific criteria. From the many different ones that are

available, one criterion has been chosen to address each of the above questions. The ease of learning question is to be measured by the number of novices who can learn to use basic facilities of the system within a given amount of time. Ease of use will be measured by the average time needed to complete a given set of basic tasks once the novice has learned to use the system. The anxiety factor is to be measured using the State-Trait Anxiety Inventory questionnaire, while user attitudes will be measured with an appropriately designed Semantic Differential questionnaire.

An experiment has been designed to evaluate Etude using these criteria. The "basic tasks" will involve typing and editing a simply formatted document (in this case, a business letter). Subjects will be secretaries without previous computer experience, drawn from temporary agencies. They will learn the system through the use of an on-line tutorial, with the assistance of the experimenter. Two experimenters will be used to control for bias from this area. We expect the first set of these experiments to be completed by the fall of 1981.

2.2. Pagination and Layout

Sandor Schoichet and Brian Berkowitz have been active in extending the Etude systems to provide the kinds of functionality typically associated with a photocompositional system, particularly in the area of flexible page layout. Our goal is to package this functionality with a very simple and easy to use interface.

The goals of the Etude project have shaped the approach taken to pagination and page layout in a number of ways. To make the system easy to use, Etude's user interface is based on the idea of *labeling*. The user specifies the type of document he is creating (i.e., *letter, report,* etc.), and then labels regions of text as instances of a variety of document *components* that are associated with each document type (such as *return address, salutation, body,* etc. for a letter). Instead of requiring the user to provide detailed formative commands, Etude can format a document automatically by looking up the formatting attributes associated with each component in the appropriate document database. This approach entails the development of a declarative database language for page layout specification.

The goals of supporting the creation of high quality documents and providing for the integration of many different applications dovetail in requiring complex page layouts. The fully made-up page is the point at which the many disparate elements of a document are brought together for display or printing. These elements include not only multiple columns of text, headers and footers, captions, etc., but also non-textual material such as images, business graphics, and line drawings. An appropriate representation for the structure of a page had to be developed to support the real-time makeup of complex page structures.

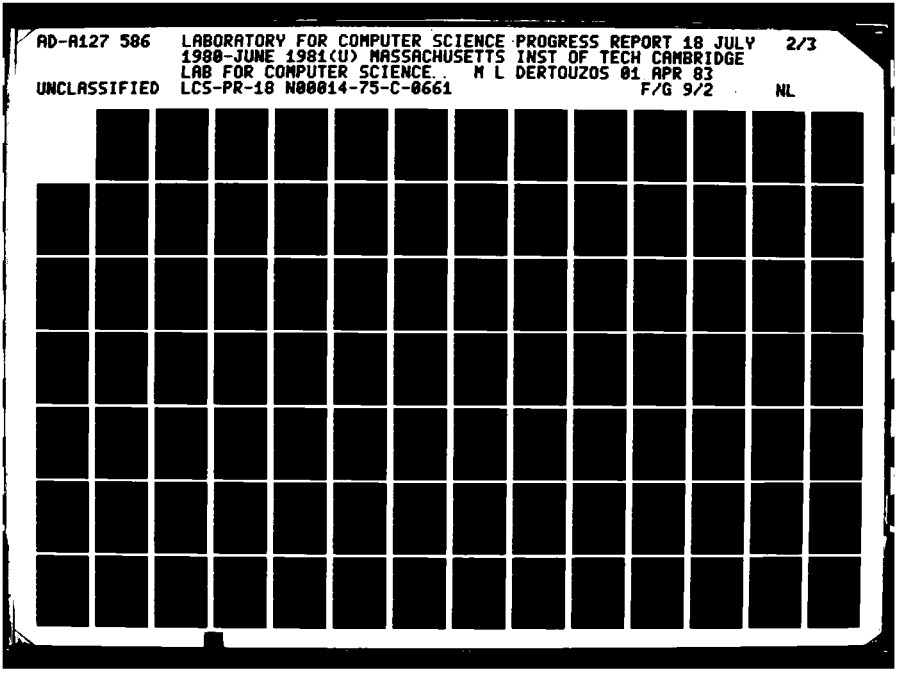
To represent the layout of a page, Etude provides a new type of constructed box, the *page box*, that allows its component boxes to be located directly in two-dimensional space. There are problems, such as page makeup, for which horizontal and vertical lists of boxes (lines and columns) in the style of TEX are not natural constructs. For example, a page in a complex document might have two or more columns of text, several cut-outs for illustrations, and a running header. Although a structure with the same appearance could be built from a complex hierarchy of line and column boxes held together with glue, it would be cumbersome to manipulate interactively if it were altered, say by the addition or deletion of an illustration.

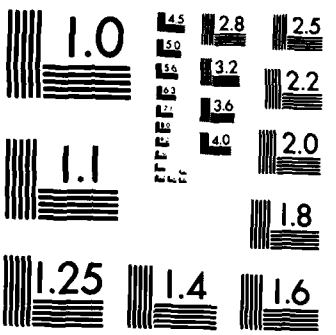
The structure of a page box is represented by a *layout* that locates a set of *containers* with respect to its upper left corner. Each container is associated with one of the columns of text that is to appear on the page. A container is represented as the union of a set of rectangles and provides the size and shape constraints under which the text of its associated column is formatted. The layout forms a simplified model of the page's overall structure that can be manipulated without concern for lower-level details. By setting the width and the horizontal position of each line appropriately as a column is composed, the outline of the column's text can be made to conform to any shape that its container may assume.

The complete text of an Etude document is not stored in a single text chain, but is broken up into a number of *subdocuments*. A subdocument is a piece of text (such as a *header* or *caption*) that has no sequential or containment relationships with other parts of the document. Subdocuments model the independence of such document components as the fields in a form or the stories on a newspaper page; they are related only by their spatial arrangement.

The component columns of a page box may come from any subdocument. On a simple page, one column may come from the header subdocument and another from the body text subdocument. The page box will have a pair of begin and end swap markers in the text chain of each subdocument that has a portion of its text appearing on the page. The fact that several columns on a page may belong to a single subdocument, with the text flowing between them, is represented by grouping a layout's containers into *container sequences*. Each container sequence is associated with a single subdocument.

In addition to *text* containers, *image*, *line art*, and *table* containers will be made available in Etude. Each of these container types will be associated with objects that can have their own unique internal representations, designed to be natural for the manipulations that will be applied to them. All that is required for such a box to be incorporated into the layout of a page is that it provide the standard small set of box attributes and operations. In this manner, Etude allows a variety of different data structures to be cleanly integrated into a single overall document representation.


AD-A127 586
LABORATORY FOR COMPUTER SCIENCE PROGRESS REPORT 18 JULY 1980-JUNE 1981(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE. M L DERTOUZOS 01 APR 83
2/3
UNCLASSIFIED
LCS-PR-18 N00014-75-C-0661
F/G 9/2
NL



The first step in developing a detailed layout specification language has been to categorize the types of documents that an office would be interested in producing, to examine a range of sample documents, and to survey the production methods used in the book publishing industry. We have examined user manuals, newsletters, and technical reports from several different types of businesses. The language we have developed allows layout information to be described at two different levels. At one level the layout of pages and spreads can be described by providing actual layouts to be used when putting together pages. At the second level the document components, e.g., footnotes, paragraphs, and sections, can have design formats associated with them describing how they should be positioned on the page.

The specification of page layout in Etude is handled by introducing the notion of *layout grids* into the database language. As the name implies, a layout grid is fundamentally a collection of horizontal and vertical lines. It's usefulness for page layout comes when the proportions of the *units* defined by the interstices of the grid are designed to work together with the functional requirements of a particular document type and its dominant typeface. The grid method is becoming widely used in the layout of complex documents such as annual reports, government documents, magazines, and newspapers. Use of a grid for establishing the layout of a family of documents, such as reports from a particular department, imparts a coherent "look" to all the documents while still allowing for considerable flexibility in the detailed appearance of each.

The layout of subdocuments on the page is specified by defining the container sequence into which their text is to be formatted. The size, shape, and location of each container is given by the union of a set of grid units. Each subdocument is additionally defined to be either *flowing* between corresponding container sequences on each page, *blocked* within its container sequence on a single page, or *running*, meaning that a blocked container sequence and its text is to be replicated on each page.

Grids also ease the problem of specifying the layout of non-textual material, such as illustrations and tables. The modules into which the page is divided provide the structure against which such *inserts* can be located and appropriately sized.

Definition of inserts in the document database, along with the other document components, allows the basic user interface mechanism of labeling to be straightforwardly extended. Facilities are provided for labeling a cursor or a region of text as an *anchor*, and then *attaching* an insert to it. The user may select any of the insert types defined in the document database (such as *2x2 picture* for a small illustration, *full page table*, etc.). The database entry for each insert specifies a set of grid units whose union will yield the insert's size, shape, location on the page, and allowable distance from its anchor. The definition of 2x2 picture, for example, might

specify that the insert is to be one grid unit in area (assuming 2" x 2" grid units), is located in the upper right corner of the page, and must appear on the same page as its anchor.

Different page layouts can be supplied for different types of pages that may appear in the document, such as chapter opening and closing pages, table of contents pages, and index pages. A library of layouts can be constructed to show preferable placement of objects positioned at the system's discretion, such as floating figures.

The language also allows certain layout characteristics of document components (e.g., paragraphs, sections, footnotes) to be specified. The component format indicates where the component should be placed with respect to other components in the document (e.g., footnotes should be placed in a footnote container at the end of the chapter) or with respect to the layout of the page (e.g., a figure should appear at the top of the current page). Another important layout parameter is if and where an object can be broken between column or page boundaries. Widow and orphan control rules that describe how objects may be broken across column and page boundaries are quite general and will allow specifications of the form: *a list cannot break across a column unless two or more list entries start in the first column.*

The layout language is also used to describe the appearance of cross-referencing structures such as cross references in text, page heads and feet, indices, and tables of contents. The document production system automatically numbers document components (e.g., chapters and sections) and keeps track of the page numbers. Using this information, the system composes the cross-reference structures described above based on their layout specifications.

In order to verify that our language allows the important layout characteristics of office documents to be described, we have begun to examine detailed document design specifications used by corporations and federal government agencies. We will also begin discussing our approach to specifying document page layout with book designers in order to determine what aspects of layout they consider most important and should therefore be part of our language.

Parallel with this investigation, we will begin implementing algorithms to produce fully made up pages according to the specifications provided by the language. The page layout algorithm must pick an appropriate layout for the page, place text on the page, leave space for figures, and determine appropriate places in the text for column and page breaks. In doing so, the algorithm will use design preferences specified in the layout language to determine how a page should be composed. In some cases it will not be possible to meet all the design criteria specified in the layout language (e.g., it may not be possible to place a figure on a page, balance the

columns of the page, and prevent an unfortunate page break such as ending a paragraph on the first line of the next page, all at the same time). The layout language will allow the document designer to specify which goals are most important. This is done by indicating a penalty that should be assessed for failing to achieve each goal. The page makeup algorithm will use these penalties to evaluate how "bad" a particular layout is, and select the layout with the lowest "badness" rating.

We do not expect the layout algorithm to always produce layouts that are acceptable to the author. Therefore, we will build a facility to allow the author to make incremental changes to the layout of a page and to see the results of these changes immediately. An important aspect of such a facility is to allow the user to indicate *why* he is making these alterations, and to include this information in the design specification for the document. For example, if the user, when scanning the fully paginated version of the document, notices a page with a section starting more than half way down a column and decides that he only wants sections to start in the top half of a column, he should be able to issue a command to start the section in a new column. The system should then guide the user so that he can easily indicate that he does not want any sections to start more than halfway down a column. Thus, an important aspect of an interactive page makeup system is the ability to specify design changes by altering the pages composed by the system. We hope to extend this facility to allow document formats to be composed by example.

2.3. Ecole

The purpose of the original Etude project was to explore user interface and implementation issues associated with an interactive text editor and formatter. Having gained this experience, we have begun to explore the issues in designing and building an integrated office workstation. One component of this workstation, of course, is a text editing and formatting system.

We have also begun the design of several other fundamental components of the workstation. These include the *graphics* system, which lets users create and modify a graphical presentation of information; the *database* system, which will let users store and retrieve entities or *objects* they deal with in the course of their work, and a *communications* mechanism, which allows users to transfer objects between users and workstations. Using these fundamental components, we will build other systems. An electronic mail system, for example, would use the text system to create, edit, and read messages, the communications mechanism to send and receive messages, and the database system to store and retrieve messages.

The goal of this project is to build an *integrated* office workstation. We recognize that there are two distinct problems of integration:

1) **Integration at the user level.** A workstation with many different systems (e.g., text, database, mail) can easily confuse the user with a multiplicity of commands. We alleviate this problem by providing consistency at the user's command level. Once he has learned a set of commands for a particular system, we build on that knowledge and make those commands perform the same or similar operations in another system. This consistency is especially important for the *user aid* functions, such as **help, menu,** and **undo.** Each system that is added to the workstation will be required to support these functions.

2) **Integration at the system level.** The representation of data objects used by the various systems should be as consistent and uniform as possible. This lets the user have documents that contain text, graphics, and images intermixed. The database can store such documents, and the user can retrieve either the entire document, or only the components in which he is interested. Integration at this level also makes it easier for systems programmers to add new systems to the workstation. For example, as described above, an electronic mail system could be brought up using the existing text and database systems. Both the text system and the database system would recognize and use the structure of a message, as defined by the mail system. With this approach, not only is the time and effort required to build a new system reduced, but uniformity and consistency between systems is encouraged.

In order to achieve this integration of different components, we have designed a system architecture that will provide a set of fundamental facilities (such as screen management and user interface handling) that will be used by each application system, and that is based on a universal document representation that accommodates multiple modes of information. This architecture is known as Ecole.

Document Structure: The users of the workstation and its various subsystems all manipulate *objects*. A *document* is the general object that contains other objects. Etude had only textual documents; these documents contained text objects, such as sections, which in turn contained other text objects, such as paragraphs. Richard Ilson has been extending the concept of *document* to allow documents to contain *arbitrary* objects.

In Etude the document structure hierarchy was implemented with each object "pointing" to its containing objects as well as to its "parent," the object containing it. This approach does not support *sharing* of objects, which we allow in Ecole. For example, a **chapter, paragraph,** or **figure** may be a component of more than one document, and thus will not have a unique "parent object." Many operations, however, require the system to know an object's location in a particular document's hierarchy; Ecole keeps this information in a structure called a *path*.

A *path* to an object is a sequence of objects from a *root* object to the object of interest, listing all the objects in between. A path to a character in a report, for example, might look like:

```
Report
Chapter-4
Section-8
Paragraph-5
Character-17
```

None of the objects in the above sequence has information about containment, thus allowing these objects to be *shared* with other documents—but the path does give information of the nesting structure in a particular hierarchy.

Each object in the path may be identified in one of two ways. Suppose we need to identify a particular paragraph within a section.

1) We may point directly to the paragraph *object*.

2) We may use an *index* into the section to locate the particular paragraph.

We can point directly to the paragraph only if the paragraph is a unique object in the system. This is generally true for paragraphs, but consider identifying a character within a paragraph. Characters differ from paragraphs in that they are not *structured* objects; a structured object may be uniquely identified by pointing to the object, while an unstructured object may not be uniquely identified this way. For example, pointing to an "a" within a paragraph does not uniquely identify a particular character within the paragraph. (We could make characters be structured objects—in fact this was done in Etude I—but the extra storage overhead for this approach makes it undesirable.)

Another way to point to a paragraph within a section is with an *index* into the section. An *index* is an integer that uniquely specifies an object within another object. (An index is not necessarily the logical offset of the contained object; the fifth paragraph in a section does not necessarily have an index of "five." This is true because objects may store their containing objects in whatever way is most efficient and convenient, and it may not be best for an object to maintain the index as a logical offset; see the section on the implementation of text objects.)

Indices are a means to point to unstructured objects, such as characters. They may also be used to point to structured objects. Indices are a more volatile way of identifying an object than pointing to the actual object. If one has an index to a paragraph in a section, and then another paragraph is inserted before it in the section, then the original index may not point to the original paragraph. For this reason we provide an "updating index," known as a *mark*. A mark is an index that is

guaranteed to keep pointing at a particular object within a containing object, independent of any changes made within the containing object (except for the removal of the original object). Marks are less efficient than plain indices; they are structured objects (unlike indices, which are just integers), and the containing object must update all marks to objects when other objects are added and removed from within it.

In summary, a path consists of a *root object*, and a sequence of objects, indices, or marks, that specify the path from the root object, and identify each of the objects down to the final object, identified by the last object, index, or mark.

In order to make changes to the document, such as inserting a paragraph or character, one needs to have both the object in which the change will be made and the location—the index—within the object where the change will be made. To insert a paragraph in a section, one needs the section and an index within it; similarly, to insert a character into a paragraph, one needs the paragraph and an index into it. Actually, we need both the object and index even to move forward or backward within the object.

Cursors are used to make changes to and move around the document. A cursor to a character in a report consists of a list, object-index pairs and looks like (in part):

Objects	Indices
Paragraph	17
Section	5
Chapter	8
Report	4

The cursor is pointing at index position "17" within the paragraph object. Index position "5" in the section is the paragraph object. Similarly, index position "8" in the chapter is the section object, and index position 4 within the report—the root object—is the chapter object.

A cursor is similar to a path in that it specifies the path to an object from some root object, but it contains both each object and each index in the path. We can always derive a cursor from a path. If the path consists of indices, these indices specify the objects in the path. If the path consists of objects, we can determine the index for each object by searching through the containing object.

Actually, there is an additional complexity. What we have described only locates an object with the document's internal (editorial) structure. Each object in a document also has a location in the document's *outward appearance*.

The *outward appearance* is a second hierarchical structure associated with the document. A typical outward appearance has several pages; each page contains

several columns, each column contains several lines, and each line contains several characters. The outward appearance of a document is composed automatically by the Etude system.

Just as a path can point to an object by means of a sequence of objects or indices in the hierarchy of a document's internal structure, a path may also point to an object by means of a sequence of objects or indices in a document's outward appearance hierarchy. For example, a path to a line in the outward appearance would look like:

```
Report
Page-7
Column-2
Line-51
```

In order to edit and move around a document, one needs to know, in general, a position in both the document's internal structure and its outward appearance. Therefore, we include all this information in a cursor:

INTERNAL STRUCTURE		OUTWARD APPEARANCE	
Objects	Indices	Objects	Indices
Paragraph	17	Line	14
Section	6	Column	51
Chapter	8	Page	2
Report	4	Report	7

The internal structure objects and indices are the same as before. We also locate the character in the outward appearance by specifying its index ("14") within the line, the line object's index within the column ("51"), the column object's index within the page ("2"), and the page object's index within the report ("7").

When an operation is performed with a cursor, either moving it or using it to edit the document, we need to update both the internal structure and outward appearance object-index pairs in the cursor. The algorithms to do this are not simple, and will not be detailed here.

Whenever a person is editing a document he is dealing with the document structure and one particular outward appearance associated with that document. A document structure, however, may have several outward appearances associated with it. For example, there might be one outward appearance of a document that describes how it prints on an electronic printer, and another outward appearance that describes how the same document prints on a daisy-wheel printer. Multiple outward appearances do not complicate the Etude document structure as described above because the user only works with a single outward appearance of his document at one time. (Different outward appearances do *share* the same

document structure; this is another reason why the document structure of Etude is inadequate.)

Implementation of Text Objects: Text objects may contain either characters of text, other objects, or both characters and objects. How the components of a text object are stored depends on what kind of components are in the text object.

If the text object contains only other objects, but not text (such as a section that contains only paragraphs), then the components are stored in a vector (array). The index to a component is simply the component's offset in the vector. When a component (that is not a character) is inserted or deleted, a new vector is created and the appropriate elements copied from the old vector.

If the text object contains only text characters then the characters are stored in a string with a hole or "gap." The index to a character is the character's offset in the string (the absolute offset, including the gap). When a character is inserted, the gap is moved to the point of insertion (if necessary), the character is inserted in the gap, and the starting offset of the gap is incremented. When a character is deleted, the gap is moved to the deletion point (if necessary), and the starting offset of the gap is decremented. As long as insertions and deletions occur at the same point in the text object, which is the usual case, the gap need not be moved, thus making changes very efficient.

If the text object contains text characters intermixed with other objects, then the text is stored as above, and the other objects are stored in a vector. A special "dead character" is used to indicate each component object's position in the string of the containing object. Thus, the index of an object is the offset of its corresponding dead character in the string. (The correspondence between the dead characters and the actual component objects is determined by searching through the string, counting the occurrences of dead characters, and using that number as on offset into the vector of component objects.)

As components are added and removed from a text object the indices of the components within the containing object may change. If we need to point to a particular character in the text object, independent of other objects being added or removed, we get a *mark* to the character from the text object. A mark is a vector that contains the index of the character. As changes are made within the text object that change the character's index, the text object updates the index in the mark to reflect the new location. In particular, this happens to a region of marks within the text object when the gap is moved. Note, however, that as long as insertions and deletions are made at a single point in the text object, then no marks need to be updated.

User Interface Handling: The Ecole user interface must deal with a variety of subsystems, allowing each to have a friendly user interface with a minimum of programming. The requirements of a friendly user interface are like Etude's: the user should be able to ask for help and menus at any point, to cancel any command, and to undo a command if it was a mistake. Tim Anderson has been developing a generalized facility to support all user interactions with the workstation.

Both the variety of subsystems dealt with and the required level of friendliness mandate an interface that is almost completely table-driven. The basic data structure of the interface is the command table, which specifies the syntax's of a set of commands. For example, the command table for Etude might have an entry:

move region cursor

In the original implementation of Etude, **region** and **cursor** were implemented by rather complicated programs; this made it very difficult for help and menus to be provided. In Ecole, the syntax of **region** and **cursor** is described as a network found in another table. This approach permits general-purpose help routines to be written: they need only examine the command parser's data structures to determine where in the command the user is, what options he has, and so on.

Ecole may have several command tables in use at any given time. For example, there might be a set of system-wide commands; another set of commands specific to the current subsystem (Etude, say); and a third set used to handle some special case. If the user is editing a graph as part of an Etude document, then he should have access both to Etude commands and to the graph system's commands, with the graph commands taking priority.

The structure of the user interface will not require that the subsystem implementor define a command table containing any commands that he wishes to implement. The **move** command is common to many subsystems; all subsystems should use the same syntax for it. The only difference is that different code needs to be executed for each subsystem.

The user interface, in addition to parsing commands, supervises their execution. The execution of a command is not directly related to the command table the command came from: although the **move** command might be defined in the system-wide command table, subsystem-specific code will be called to execute it. Let us consider the execution of the following command, assuming it was typed while the user was editing a graph as part of an Etude document:

delete previous paragraph

It is not likely that a graph will contain paragraphs, but it is quite likely that the graph system wants to have a **delete** command with the same syntax that Etude's

delete command has. The user interface, then, must first ask the graph system to evaluate the expression **previous paragraph**. Since it can't, the interface then asks Etude to do so. Etude can, so this is really an Etude command, though it was typed inside a graph, using a system-wide command table. The user interface will therefore call the Etude **move** routine with the region returned by Etude's evaluation of **previous paragraph**.

The user interface can therefore be thought of as an interpreter for command tables; it defines the control structure of any Ecole subsystem by defining the calls that are made to the subsystem's code. Functions such as help and menu generally can be provided by allowing the user interface to examine its current state; undo requires relatively complicated interactions between the user interface, which is storing the data, and the subsystem, which is using it.

Display Management: Larry Rosenstein has been developing the Ecole display manager, whose function it is to coordinate output from different subsystems. This coordination is needed because the workstation's video screen is a scarce resource which must be shared by the different applications programs, as well as by different parts of individual programs. Information must be presented in a consistent and logical manner; a subsystem is useless if the user cannot understand its output.

The display manager is also important to the application programmer. Only low-level operations, such as displaying characters and vector, are built into the display. There are no mechanisms, for example, for organizing information on the screen. The display manager provides a set of high-level data types and utility functions, which simplify the use of the screen by programs. Two of its important parts are its implementations of a *window* data type and an *incremental redisplay* mechanism.

Windows serve two functions. First, they organize information on the screen. Related pieces of information can be grouped together in a window; each group can then be moved or updated as a unit. Windows behave like pieces of paper on a desk; if two windows overlap, one obscures part of the other. Just as it is easy to rearrange papers on a desk, it is easy to rearrange windows on the screen, in order to display any desired information.

The second function of windows is to limit the part of the screen a subsystem can use for its output. This feature helps to support subsystem integration. For example, if the user wishes to make a database query while editing a document, it is likely that the query will involve information contained in the document. Therefore, the user will want to keep that information visible on the screen. Using windows, the output from the database subsystem can be confined to a small window, which the user positions in a convenient place.

The other important part of the display manager, besides windows, is its *incremental redisplay* mechanism. An incremental redisplay mechanism reduces the time needed to display a data structure. Consider the problem of editing and displaying a text document. The simplest method for displaying it would be to output its entire content each time it is changed, which could take several seconds. Most commands, however, only change a small part of the document; inserting or deleting a character, for instance, usually affects a single line.

If the content of a document does not change very much, then its new appearance on the screen will not be very different from its old appearance. Completely redisplaying the document each time does not, however, take advantage of this fact. An improved redisplay strategy would keep track of the content and position of each line displayed on the screen, and fully redisplay a line only if either of these attributes changes. This is the underlying concept of incremental redisplay; the work done each time to redisplay a document should be proportional to the degree to which it changes.

Each application has certain inherent information which the user manipulates. The programmer's task is to: (1) define a data structure to represent that information in the computer; (2) provide commands for manipulating the representation; and (3) display the results. There are usually many ways to represent a particular piece of information in the computer; part of the art of programming involves weighing the memory and efficiency considerations, and designing a data structure appropriate to the situation.

Similarly, the programmer must define the *outward appearance* of the data structure, which is its representation on the screen. This task is analogous to the problem of presenting statistical information. Raw data is usually not very revealing; the same information presented pictorially, however, is much more comprehensible. There are several ways to present the same data; for example, with a graph, pie chart, or histogram.

As in the case of statistics, a data structure in the computer is only a series of numbers which cannot be meaningfully displayed on the screen. The information must be converted into characters, lines, etc., which can be displayed; the outward appearance defines this conversion. Like statistics, the same data structure can have several outward appearances. A document, for instance, can be fully displayed, or the section names can be presented and displayed in outline form.

To define the outward appearance of a data structure, the programmer creates a *displayable object* (disp-obj). The disp-obj contains the data structure and a *redisplay function*, which invokes the necessary output operations to display it. By associating the redisplay function with the disp-obj, rather than the data structure, it

is possible to display two views of the same information simultaneously. The disp-obj also insulates the rest of the display manager from the representation of the information.

Another purpose of the displayable object is to define the parts of the outward appearance that will appear on the screen. Usually the entire outward appearance is larger than the screen area in which it is to be displayed; in Etude, for example, only the part of the document near the cursor is "interesting" to the user.

Once his displayable objects are defined, the programmer arranges them on the screen, using *windows*. Although he can put them all in a single window, usually a more structured arrangement is desirable; related information can be grouped together, and moved and updated as a unit.

Windows can be organized into a hierarchical structure. A window can contain both displayable objects and other windows. Each object contained in a window can be positioned in three dimensions within its parent. In addition to its horizontal (x) and vertical (y) positions, the depth (z position) of an object, relative to its siblings, is important because it obscures objects underneath it.

The display manager automatically keeps track of the visible and empty parts of the window. The former is the part of the window within its parent's visible part and not covered by any sibling; the latter is the piece of the visible part not covered by any child.

There is one window built into the display manager; the *physical window*, which has the same shape as the physical screen. Before a window or disp-obj can appear on the screen, it must have the physical window as an "ancestor." If this is the case, the window can be redisplayed using a function provided by the display manager. This function first redisplays all the objects contained in the window, and then erases the window's empty part.

Ultimately, a data structure's outward appearance is displayed on the screen. This process involves generating the appropriate output operations from the information in the data structure. In the case of a text document, this involves outputting characters in different fonts at the correct positions of the screen. A graphics editor, however, would need additional operations to display lines, curves, and raster-scanned images. Finally, the implementation of incremental redisplay uses general bitmap operations to erase and move arrays of pixels on the screen.

Functions do not output directly to the physical screen. Instead, output is handled by *screen* objects. A screen object acts like an idealized output device, which occupies a part of the physical screen. Any kind of operation that can be performed on the physical screen can be performed on a screen object. In addition, the screen

object insures that nothing is printed outside of its boundaries. This feature allows several subsystems to share the screen without conflicting with one another.

In most cases, the programmer will want to use an *incremental redisplay* mechanism to reduce the time needed to display his data structures. Completely displaying the data structure after each command is inefficient, because most commands only change a small part of the data structure.

Ideally, the work done to display a data structure should be proportional to the degree to which it was changed. This is the concept behind incremental redisplay. Extra processing time is used to plan a more efficient redisplay strategy, in the expectation of reducing the overall redisplay time.

It is not practical, however, to use an optimal redisplay mechanism, for two reasons. First, the time needed to develop an optimal plan would outweigh the time it would save; the cost of executing a plan includes not only the cost of the output operations, but also the cost of constructing the plan. The brute force redisplay algorithm has no planning cost, but a high output cost; an optimal algorithm is just the opposite. Unfortunately, it is not possible to calculate the costs of various redisplay strategies in order to choose the best.

The second reason is one of human factors. Users will be distracted if a large part of the screen changes after every command, especially if the command only makes a small change. In addition, they would prefer that the screen is updated consistently each time; for example, always from top to bottom. In this way, they can get some idea of how much longer the redisplay process will take.

For these reasons, the most appropriate incremental redisplay strategy is one which handles the common cases well, and ignores more unlikely cases. For example, the insertion and deletion of individual characters occurs very often, and should be handle by the incremental redisplay mechanism. An operation that transposes every successive pairs of lines is not common, however, and the incremental redisplay mechanism should not be expected to handled this case efficiently.

Another consideration in the design of an incremental redisplay mechanism is the division of work between the subsystem and the display manager. At one extreme, the subsystem could do all the work. It knows exactly what part of the data structure was changed; the display manager would simply provide output operations for moving information on the screen.

The disadvantage of this approach is that each subsystem programmer has to implement his own incremental redisplay mechanism. The display manager, however, should be able to provide more assistance to the programmer because there are common aspects to each incremental redisplay mechanism.

The exact mechanism that will be used in the Ecole display manager has not yet been determined. Like other parts of the display manager, it should be applicable to a variety of subsystems, and there should be enough flexibility so that programmers can implement their own incremental redisplay mechanisms if they choose.

An initial version of the display manager, without an incremental redisplay mechanism, was implemented in CLU. One feature of this implementation was that windows did not have to be rectangular; their shape could be any area composed of a union of rectangles. The problem of overlapping windows was solved with a general data type for manipulating such areas.

In addition to the basic display manager described above, data structures and associated redisplay routines were implemented to test the display manager. One type of data structure was a text buffer that the user could edit.

The CLU implementation of the display manager was used in a prototype graphics editor and in the calendar system being developed by the Office Automation Group. Its use by other programmers helped to point out some shortcomings in the design and implementation. At the present time, a new implementation of the display manager is being built which will be used in the next version of Etude. This version will be written in MDL and will include an incremental redisplay mechanism.

2.4. Office Database Management

The field of data processing has grown up over the past decades in response to the needs of business to automate some of their costly but simple functions. Early applications focused on well-understood tasks such as payroll and accounts receivable. Database technology was developed as a technique that would support this applications development process.

The field of data processing is now moving into more complex applications areas in the new field of office automation. Our approach to this field is that it is not different from data processing, but, rather, it is the next logical step in the development of business application technology. Consequently, we feel that database technology has a lot to offer office automation. Ed Gilbert and Stan Zdonik have been exploring how the paradigm of database management has to be extended in the office environment.

We believe that office applications have characteristics that distinguish them from conventional data processing applications. If we look at the applications that data processing has addressed, we see the following characteristics:

1) They tend to be *highly structured* in the sense that the entire process

that is to be automated can be described in detail. The criteria on which the decisions are based can be specified in sufficient detail such that the decisions can be made automatically. In a payroll application, the deduction for medical insurance can be determined by the kind of coverage that the employee has elected and the plan that is providing the coverage.

2) The tasks that are being automated tend to be *repetitive*. A payroll program may run once a week doing the same work that was done last week.

3) The applications are *formal* which means that they tend to follow a well-defined procedure with very few exceptions. The main-line processing is the rule. Almost all checks cut by the payroll program have a standard salary with a set of standard deductions.

4) There is a *high transaction volume*. The number of interactions with the system in a given time period is large even though the the types of interaction are fairly uniform. The payroll system will cut many checks in any large corporation, while the process involved for each check is the same.

In contrast to the conventional set of data processing applications, office applications have a somewhat different set of characteristics. It is as a result of these differences that we believe that the database management systems that will successfully support them will be different from the state of the art in database technology today. These distinguishing characteristics are given below:

1) We would characterize most office procedures as *semi-structured*. By this we mean that they are a combination of structured activity as described above interleaved with unstructured activity. An unstructured activity is one that can not be adequately described to be meaningfully automated. An example of this type of application is a college admissions office. The processing of an application consists of making sure that the proper forms are sent out to the appropriate parties on time, something that can be described and, therefore, automated, and the making of decisions such as who should interview an applicant which depends on many subjective criteria and, therefore, cannot be automated.

2) *Occasional access to a small number of objects* is more common in an office than routine access to large numbers of objects as in the data processing environment. An admissions officer will want to see the

recommendations of a particular applicant in response to some question that has come up. It is hard to see a case in which it would be appropriate to process all letters of recommendation at the same time in batch mode.

3) The applications that will be run in an office on a given day is *less predictable* than in a data processing environment. Ad hoc use of the applications software is more common than use that can be planned a priori. The admissions office typically does not know that it will need to look at three letters of reference and four grade summaries on a given day. Instead, an admissions officer will access this information only when the need arises based on some unpredictable stimuli.

4) There is a much *lower transaction volume* in an office than in a data processing environment, even though these office transactions are less uniform. There may be more transaction types used in given day, but the total volume will be less.

5) For formatted data, there are *fewer records* and these records will be *less uniform* than in the data processing case.

It is also important to realize that in the office environment, one is interested in small-scale, office-specific applications for which conventional development processes are inappropriate. The notion of creating applications systems by hiring a team of programmers and a management staff to direct them is not feasible in a small office. Instead, there must be ways for the office worker to design and build the software that is required.

As a result of the differences cited above, we feel that OA database management must possess certain characteristics that are not present in the database management systems of today. Some of those new features are listed below:

1) The database system must be able to support the creation of *highly interactive programs*. The nature of office applications is such that the programs that implement them must interact heavily with their users as well as with other programs in the system.

2) The DBMS must be able to deal with *multiple modes of data*. In the office environment, there are many different kinds of objects that must be treated in a uniform manner. Some examples of object types that must be handled are documents, graphics, calendars, and tables. Although one could possibly shoe-horn these data objects into a records based system, the objects are not inherently records oriented. There is a need to handle records too.

3) The system should be able to deal with *non-formatted data*. By this we mean that the interpretation comes from the person using the system. In a conventional database environment, a value of 30K might be stored as a value of an attribute named *salary*, meaning that the employee's salary was 30K dollars. The interpretation of the value was provided by the name of the attribute to which it was attached. The value of a paragraph component of a document is a long text string the interpretation of this string is provided by the reader.

4) In an office database environment there will be no DBA as there is in most large-scale databases of today. The job of organizing, documenting, and maintaining the data must be done by the office worker with a great deal of help from the system.

5) Since the old model of applications development by a team of highly skilled programmers cannot apply in an environment of limited resources, there will be a need for more application development done by the end-user. There must be tools that accompany the database system to assist users in this.

6) There will be more use of data for operational decision making by the office worker. There must be a convenient facility for the formulation of queries that can be used by the office worker to retrieve objects that combine the many kinds of data that will be stored in the database.

7) The office environment will be inherently distributed (decentralized). Workers at different work stations will be creating objects that fit their own model of the world. There must be a facility for coordinating their efforts such that they can share the objects that they produce.

An office database management system can contribute to the integration of the subsystems that are present in an office workstation. Examples of such subsystems are a text editor/justifier, a graphics system, a table package, a calendar and scheduling system, and an electronic mail system. If we examine these subsystems, we see that each of them is concerned with a particular type of object. The subsystems provide the operations for creating and modifying these objects. The database system can provide a uniform view of these object types by providing semantic structures and operations that are generally useful for modeling these objects.

Stan Zdonik has been working on the design of an office data model that would be the basis for a general *office object support system*. The data model includes facilities that are not present in a conventional database management system. These features arise out of the unique characteristics of office applications systems.

The office object support system will be accessible by all the subsystems as well as by a query language module that provides a top-level user interface to the object semantics of the data model. Some of the features of this data model that distinguish it from conventional data models are:

1) This data model is object-oriented as opposed to records-oriented. In a conventional database system, the basic data structure is a record with fixed structure and fixed length fields. This is adequate in an environment in which the data structures are intended to capture short descriptions of various attributes of objects in the real-world. In the office environment, however, we are interested in dealing with classes of objects that are actually stored entirely in the machine. A document stored in an office database is not a description of some real-world entity, but, rather, is an entity in its own right.

   Here, we would like to support these objects which can have variable structure. For example, a document might or might not have sections for some of its chapters. The data model should be able to express this directly as opposed to having a set of section fields for each chapter with value *null*. The *components* of an object are other objects that can be of arbitrary size.

2) The objects should allow *sharing* of components. It often happens that part of one office object (e.g., a document) is another object that has existence of its own (e.g., a graph). In fact, the graph might be maintained by a different person than the worker who is creating and maintaining the document. It would be convenient to have any change to the graph object to be automatically reflected in the document object. This can be accomplished by actual sharing of the common object (i.e., the graph).

3) The office environment is characterized by continual modification of the objects in the database. It is essential that there be a way to keep track of the history of a given object. In a conventional file system (e.g., TENEX), the entire object that is being modified is copied to create a new version. In the office data model, *version control* can be done at the component level. That is, when a component of a structured object is changed, only that component needs to be copied as long as all objects that contain it as a component are aware of the change.

4) The *outward appearance* of an object is an object in its own right. It has structure that is different from the structure of the object on which it is

based (i.e., the logical object). The outward appearance object (i.e., the physical object) that corresponds to a document object would consist of components such as pages, columns, lines, and characters. There may be several outward appearance objects for a given logical object, one for each output device or format. The outward appearance could provide valuable information to support queries based on visual queues. The user might like to see all reports that have been produced in the last month that had a graph in the upper right-hand corner of one of the first two pages. This is a search through a set of outward appearance objects

5) The office data model should provide an *easy to alter schema definition*. A user who is creating a report might want to create an appendix that has not been defined as a legitimate component for this document type. There should be a facility for defining alternative conceptualizations for an object type.

6) It is often useful to be able to create *aggregations of arbitrary object types* in the same way that one would do with a file folder.

7) One problem with unformatted data is that the interpretation of that information is not captured in the system. An approach that seems to have some utility in trying to capture some interpretation of the unformatted data is to allow users to attach descriptive material to any object. This material could be in the form of a short textual comment or it could be in the form of a formatted record. The ability to associate comments with an object is a useful facility independent of the object type.

8) An area that seems to place many requirements on the data model is the support of *cooperative processing*. By this we mean the coordination of a single task, possibly centering around a single object, by many workers. An example of this type of activity would be joint document production. Some of the facilities that would be useful for this are given below:

    a) There should be a facility to support *editorial control* over the object. That is, each person involved should be able to include comments that are analogous to "red-marks" that one might write in the margin of a document. There must also be a distinguishable way to include alternative phrasings for existing sentences or sentence fragments of the document.

b) *Concurrency* is an important issue in an environment in which many people are working on the same object. Concurrency should be supported at the component level. If one person is changing a paragraph, others should be prohibited from changing it. Several people, however, could be attaching comments to a paragraph at the same time.

c) In the process of joint authorship, workers would at times require their own *private copies* of an object. They could then work on this copy independently of the others and, at some time in the future, could *publish* their version back into the main stream.

9) When an object of a given type is modified in a given way, a message should be sent to the custodian of that particular object instance. This is an example of *active objects*, objects that notice when something important has happened and respond to that occurrence by performing a specified action. This facility is very useful to organize the process of alerting users of the state of the office data.

2.5. Integrating Non-textual Information in a Document Preparation System

One objective in an advanced workstation is to produce non-textual material along with textual material in documents. There is hardly a document that does not contain some kind of a table, diagram, picture, or graph. Even some purely text objects that have a rigid structure (as opposed to the body of text that may be flowed) such as a table of contents or an equation may be more appropriate to set by graphical means of spatial relationships and coordinates.

Traditional document preparation systems have treated textual and non-textual material as separate parts of the same document which get assembled in the last stage of preparation by a cut and paste method. Recent commercial typesetting systems and some document preparation systems used in the research community automatize only the last step of integration in that the text and diagrams are merged electronically before final output.

Bahram Niamir is developing a facility to integrate textual and non-textual material at document preparation time rather than proof-reading time. An interactive real-format text editor and a (de facto real-format) graphics editor provide tight feedback in the process of preparing separate parts of the same document. Real-format integration hoists the process of parts merging into the document preparation feedback loop.

We identify the following issues in text and non-text integration:

Two dimensional layout: Algorithms have to be developed that efficiently specify a layout for the document with areas set for tables, diagrams, and figures (which we will collectively refer to as graphics) that are tied in spatially with the surrounding text. In addition, the layout system must have the capability to dynamically reconfigure the composition as the graphics area grows or shrinks during the editing process and convey the changes to the text redisplay process.

Re-invocation: A graphical object such as a table is itself composed of different areas, with possibly different objects in the areas. A graphical object itself has to be laid-out. Some areas such as those containing captions and lettering may best be produced by invoking the full power of the text editor on them.

Command interface: It is a recognized fact that providing a uniform command interface along system boundaries to the user will increase productivity and reduce the effort required to master the various systems. It is yet to be shown to what degree text manipulation semantics have elements in common with graphical manipulation semantics so that a uniform user interface may be developed for both.

References: We need to resolve references across system boundaries. Table numbers and figure numbers must become available to the text editor. As another example, page, section, and chapter numbers must be available when producing a table of contents.

System related issues: We need to find the appropriate way to interface the graphics editor to the screen, window, and layout systems. What are the useful functions that could be shared between the text and the graphics editors? How would a database management system be used to provide an automatic facility for storage and retrieval?

We investigated one aspect of the last issue. A table editing program was developed to create tables using the rudimentary character oriented graphics of a standard display terminal. It relied heavily on the Ecole system which acts as the resource manager and interface for a number of subsystems including the text editor. The result was encouraging, pointing out the facilities needed to interface a table editor to the document preparation system.

3. FUNCTIONAL OFFICE AUTOMATION

A major premise of our research program is that the goal of office automation is to improve the realization of the business functions that are an office's fundamental reason for being. While improved efficiency in an office's information handling tasks

may contribute to such an improved realization, it does not necessarily do so. Our approach emphasizes the development of office-specific, *functional* systems that *directly* support or automate the substantive work being performed in the office. Such systems are *integrated* collections of diverse components, which are bound together by appropriate application software. The advantages of our approach can be considerable, when contrasted with conventional task-and tool-oriented approaches: significant improvements that are measurable in real business terms, emphasis on professional rather than clerical workers.

Unfortunately, there are considerable costs associated with the implementation of such automated office systems: namely, the labor-intensive processes of analyzing existing office procedures, designing a revised system, and constructing the requisite software. Our research is aimed at ameliorating these problems. To date, we have focused on the development of an Office Analysis Methodology (OAM) and an Office Specification Language (OSL), which support the analysis of an office's operation and its description in a precise yet natural notation. These tools can be used for such purposes as documentation, training, and redesign as well as in the context of system implementation. OAM and OSL, while distinct constructions, share a common set of premises and a conceptual framework, and are becoming more closely linked. We are now engaged in extending these tools to address the complete process of functional office system construction.

A major difficulty in the process of tailoring a general system to a specific office is the amount of time and effort required to analyze the existing operations of the office. To help reduce the investment required in properly designing an automated system for a particular office, we have developed an Office Analysis Methodology (OAM), a set of techniques for studying an office's activities in a way that elicits its fundamental structure and for organizing a coherent description of its operation. The design of OAM is one that we have found to be effective, based on our experience in doing many such office studies. The methodology provides guidance in areas that range from conducting an interview to identifying a unifying structure in apparently complex office procedures.

3.1. OAM

Marvin Sirbu, Michael Hammer, Sandor Schoichet, Jay Kunin, and Juliet Sutherland have continued the development of OAM.

Our purpose in analyzing an office is to develop a description of its operation in *business terms*. Thus, our focus is not on individual forms nor on the job descriptions of individual employees; we are not concerned with preparing minutely detailed flow charts showing the disposition of every type of correspondence

handled by the office. Rather, we are interested in the *functions* that the office performs (purchasing, collecting accounts, planning) at a level of abstraction above the particulars of the current implementation. The *goals* of the office are central to our understanding of its operations. This focus reflects our interest in first *describing* current operations for the purpose of analyzing them and then *prescribing* an improved implementation.

Before an analyst approaches an office, he must be clear as to why he is studying the office: who requested the study, what the ultimate result is expected to be, how much authority the analyst has. The answers to these questions will profoundly effect the details of the interview process. For example, when the analyst has been informed by management that the purpose of this study is to determine if the organization can function without this particular office, the problems that the analyst will encounter in dealing with the personnel of the office will be quite different from those encountered when the manager has requested a study prepatory to investing in office automation equipment intended to remove some of the burden on his staff.

The first stage of an OAM study is for the analyst to meet with the office manager. One purpose of this interview is to come to an agreement with the manager on the ultimate objectives of this study. Is it an end in itself, the first step in an automation effort, or the beginning of a reorganization? Another purpose is to obtain answers to certain key questions: What is the mission of the office, its organizational context, and its key interfaces with other offices? What is the organizational structure within the office and its reporting relationships to higher management? At this stage the analyst should only try to get a map of the territory and to avoid too much detail. Key personnel involved in each of the basic procedures carried out by the office should also be identified. Finally, the analyst needs to discuss with the manager how the office staff are to be informed that the study is taking place. There are several ways to handle this, ranging from individual meetings with each staff member to a general meeting with all of the staff. The important point is that the staff be told, in some detail, that the study is occurring, and what its goals and objectives are.

The next step is to gather information about the "main line" or ideal case of each major procedure, through interviews with the personnel who are actually responsible for them. Most office activities appear to be enormously complex to the outside observer. Our view is that this complexity is not inherent in the design of the procedure itself, but rather is an artifact of in the myriad exceptions that arise in practice. By concentrating on how things are *supposed* to work, it is possible to arrive at a description that is not overly complex or obscured by detail. The important information to gather at this stage is the *purpose* of each step of the procedure, its inputs, the databases referenced, and its outputs.

In conducting an interview, the analyst should realize that most people do not

have an internal model of the structure of their own work. For this reason, the tendency of a typical office worker is to give an unorganized description of what he does, usually with too much detail in some places and not enough in others. The OAM analyst employs a model of the structure of office procedures and attempts to fit what the interviewee says into that model. The analyst also attempts to structure the interview so that no important information is missed. In our experience, interviews that last more than an hour to an hour and a half are to be avoided, since the interviewee resents losing so much time and the interviewer begins to suffer from information overload.

Wherever possible, the analyst should interview at least two people who do the same or similar parts of a procedure. This allows the analyst to corroborate the information provided by each person. The number of people who need to be interviewed in an office can range from very few to every person, depending on the variety and distribution of job types.

When the analyst has completed these initial interviews with the appropriate office personnel, he should step back from the fray and consolidate a picture of the overall operation of the office. The analyst is now in a position to identify the conceptual "objects" that are at the heart of the office's operation. The concept of this "object" is one that underlies both OAM and OSL. It is our premise that most office procedures can most effectively be described as the life history of some object, which may be tangible or intangible. Examples of such objects include a purchase, a loan, or an application. (Notes that these objects do *not* correspond to specific forms or other pieces of paper.) A typical office procedure describes the life cycle of such an object from its origination through to some completed condition. Moreover, we find that a collection of related procedures that are concerned with similar objects can often be organized in terms of of unifying structure we call a *function*.

Only when the analyst is clear about the *structure* of this work in the office should he begin to collect additional information. Before setting up additional interviews with the office staff, the analyst should write a first draft of the description of the office. This draft should be circulated to all of the people who were interviewed, including the manager, so that they will have a chance to review the sections pertaining to their work for errors or incomplete information. A second round of interviews can then be scheduled.

This second round of interviews with the operating personnel is used to collect information about the exceptions to the ideal procedures, to resolve any conflicts or inconsistencies in the earlier information, and to gather some quantitative information about the operation of the office's procedures. The structure already imposed on the operation of the office by conceptualizing it in terms of abstract objects, the procedures that manipulate them, and the functions that group the

procedures, provides guidance in looking for exceptions. This structure also assists in choosing useful measures for quantitative analysis.

At various times during both rounds of interviews, the analyst should do some first-hand observation. The principal purpose of this observation is for corroboration and hypothesis-testing, to validate the analyst's perception of the office, to ensure that the interviewees' responses have been accurate, and to identify key exceptional situations. Similarly, the analyst should conduct interviews at the person's normal place of work so as to be able to observe any interruptions that may occur. Interviewing a person at his desk also gives the analyst the opportunity to observe the physical layout of the office and to ask about any relevant equipment or files that may be nearby.

When the second round of interviews is complete, the analyst should prepare a final version of the description of the office. This description follows a standard format. It begins with a clear statement of the mission of the office in business terms. Then the organizational structure of the office is described, including its internal structure and its position in the larger organization. A description of the physical environment is also useful, since it often influences how procedures are actually carried out. The body of the report is taken up with descriptions of each separate procedure carried out by the office. These procedures are usually organized by function into chapters.

The final stage of the analysis process is to review the provisionary model of the office with the office manager. The analyst should send a copy of the final description to the manager, to give him a chance to read it before the interview. This second interview with the management is intended to accomplish a different set of purposes than the first. Most important is to validate the *intentions* behind various elements of the procedure, and to identify the more general aspects of exception handling within the office. The manager should also be able to clarify any remaining questions about the interface between the office and other groups within the organization. One last revision of the final description is made to incorporate the input from the manager, and the analysis process is complete.

During the past year, we have continued the development of OAM, principally through its application by ourselves and others to operational offices. These case studies have given us valuable insight into the strengths and weaknesses of the methodology, and have resulted in a number of important modifications to it.

Most of the studies that we performed this year were of MIT offices. Among these are:

1) The Industrial Liaison Office, responsible for managing the Industrial

Liaison Program, which helps MIT exchange technology with companies from around the world.

2) The Student Loan Office, responsible for the maintenance, billing, and collection of student loans at MIT.

1) The Office of Facilities Management Services, responsible for maintaining information about space and equipment at MIT, as well as for developing and maintaining a computerized space management system used by many organizations outside MIT.

2) The Career Planning and Placement Center, responsible for counseling students about career choices and for scheduling interviews for students with on campus recruiters from industry.

3) The Alumni Office, responsible for all activities at the Institute that involve alumni.

Each of the above studies has been interesting for a different reason. After our study of the Industrial Liaison Office (ILO) was completed, the ILO decided to purchase a computer-based support system. The office description that resulted from our study was used as the basis for the Request for Quotation for this support system. The members of our group who performed the study of the ILO have subsequently participated in the design and implementation of the support system, providing us with additional experience in the complete process of automating an office.

The study of the Student Loan Office was done by a new graduate student, who was given only the OAM and OSL documents to work from. This study showed that a novice, with very little instruction, could successfully carry out an office analysis using OAM.

The Office of Facilities Management Services has provided us with another opportunity to experience the entire process from initial analysis to final implementation, but in this case, the decision to invest in some type of office automation equipment was made before the study was begun. An additional result of the OAM study will be some recommendations about what equipment should be purchased and how it ought to be used.

The Alumni Office study is interesting in that it has provided us with experience in studying larger offices. In addition, we tried the experiment of having several people working together on one study and learned a great deal about how best to manage team work.

Partly as a result of these studies, we have extended OAM for use in large, hierarchically organized offices, and have broadened its scope to address the planning issues related to the initiation of a study. While we are now engaged in the process of developing additional methodologies for other stages in the development of an integrated office support system, OAM as a detailed office analysis technique appears to be relatively effective and stable.

3.2. OSL

Michael Hammer and Jay Kunin are continuing work on OSL, an office specification language that can be used to describe offices and their procedures in a natural way.

OSL is a formal, high-level notation that is based upon a model of office structure that has evolved in the course of our research. The structure of OSL is derived from study of a large number of office studies from which we have sought to abstract recurring concepts, actions, objects and constructs. OSL therefore represents a particular conceptual framework in which to think about and describe office work. The detailed features of OSL were described in last year's Report, and will not be repeated here. Work in the past year has concentrated on revision, enhancement, and testing of OSL and its underlying theory.

Several important premises underlie our approach to office analysis and specification, and form the basis for the development of OSL:

- There *is* structure in an office. An office exists to execute some business function(s) for the organization of which it is a part; it has a particular mission. Therefore office analysis and specification tools should be expressed at the level of business functions, rather than the level of typing, dictating, and telephoning.

- The people in an office have control over some parts of their environment and not over others. Thus a study or description of an office must recognize its connections with, and dependencies on, both the organization of which the office is a part and the world external to it.

- Office procedures do exist, and provide the structuring mechanism that organizes the individual activities of office workers. Organized sets of procedures implement the business functions of the office.

- Office procedures are *semi-structured*. That is, routine, algorithmic, processing is interspersed with the need for creative action by office

workers. This is true for clerical workers as well as for secretarial, managerial and professional personnel. Indeed, most white-collar jobs involve a combination of these types of work. Therefore office procedure descriptions are idealizations. Often, things do not go the way that they are supposed to; the procedure cannot be followed. It is then the job of the office worker to deal with the situation as best he can; he tries to make things *appear* as if they had gone right. The ability of the worker to do this depends both on his knowledge and understanding of how the procedure is *supposed* cooperate, his freedom of action, and his creativity.

- Office procedures are fundamentally *simple*. The observed complexity of office work represents historical accretions and anachronisms, as well as various exception-handling mechanisms, typically organized around a basic core of operations. This "main line" of the procedure, which is what is supposed to happen when everything goes right, must be understood before sense can be made of the variations and exceptions that form the mass of the apparent complexity of observed office procedures. Therefore analysis and description of office procedures should concentrate upon the ideal; variations and exceptions should only be examined in light of the context provided by the main line. OSL enforces this philosophy by requiring that the main line of a procedure be described separately from variations (which involve differences in processing necessary for *a priori* differences in the focal object) and exceptions (which are problems that prevent processing from continuing).

- Office procedures should be oriented around abstract objects. We believe that office automation techniques that are based upon the following, counting, and optimizing of forms or other such documents are misdirected. Focusing on more abstract entities in an office (e.g., "program", "schedule", "purchase") rather than on the artifacts of the office's current operations implementation, can force clarification of business goals and enables a procedure to be described as a sequence of desired state changes. OSL enforces this point of view by requiring every document to be identified with some other (more fundamental) entity about which the document carries information.

An OSL specification has a canonical structure that includes descriptions of both the environment of the office and of its operations. The environment is described primarily as a number of classes of *entities*, each of which has a set of *attributes*. (This approach is derived from earlier work in our group on semantic database

modeling.) OSL includes a vocabulary of built-in entity types that have well-defined semantics; an analyst describes a particular office environment by defining the entities of interest in terms of the built-in types, and adding relevant attributes to further characterize each such constructed type.

The operational part of an OSL description follows a canonical, hierarchical organization. The top level of the hierarchy is a *function*, which is concerned with managing a set of entities over time; its goal is the maintenance of these entities (called *resources*) and information about them. Examples of functions (and of the resources that they manage) include service dispatching (servicemen), television station sales (time slots), and sponsored research administration (research programs). A function specification in OSL is based upon a simple model that organizes a number of related procedures, each of which deals with some aspect of management of a resource. These include *initiating* and *terminating* procedures that deal with the creation (or recognition) of resources and the processing required when a resource is no longer of interest. Other procedures perform required processing that is triggered by external events relevant to the resource. Functions often cross office boundaries, and therefore provide a high-level means of understanding how operations in different offices relate. This model of office structure is shared with OAM.

An OSL procedure is oriented around some (usually) abstract entity, the *focal object* of the procedure. A procedure describes the active processing of its object from an initial to a final state. Its goal is the disposal of its object; when the object reaches its final state, the procedure terminates. The main-line control structure of a procedure is described by means of a state-machine-like construct, consisting of *states*, which indicate points at which processing of the object is suspended pending an external *event*, and *steps*, which describe the processing of objects in passing between states. Steps in turn are broken down into *activities*, each of which is a verb from the OSL vocabulary that describes a particular kind of processing or decision. *Timing constraints* on events provide some quantitative information about the information flow; other such information is included in a separate part of an OSL description. Also separate from the main-line are sections for the description of variations and exceptions.

The following example of OSL is a part of a description of the MIT Admissions Office used in an OSL training course. (The language revision currently being undertaken will incorporate significant syntactic changes from the version employed here.) This example is meant to give a flavor of one version of the language. The example includes the description of the undergraduate admissions procedure, and the definition (from the environment part of the description) of an application, which is the focal object of the procedure. An application is defined as being a *record*, which is one of the built-in entity types used in the construction of all environment specifications.

Although the example will not be completely understandable without detailed knowledge of the language, a few notes on OSL syntax should assist in reading it. By convention, all words in CAPITALS are names of entity types, either built-in (such as RECORD) or defined (such as APPLICATION). Names of attributes of entities are in small letters with initial capitals. The syntax of an attribute definition is <Attribute name>: <value type>, where the value type is either the kind of entity that the attribute names or a reference to the value type of another attribute.

The syntax of activity specification is <subject> <activity> <object>, where (unless otherwise specified) the subject is assumed to be the person "Responsible" for the procedure, and the object to be the focal object of the procedure. All nouns (other than specific dates) used in the operational specification are either classes or attributes, which are defined in the full environment description. When an attribute name is not specified completely (i.e., it is used without an identifying entity), it is assumed to be an attribute of the focal object of the procedure.

```
Class APPLICATION is RECORD
    Address: Refers.Address
    CB-letter: CB-LETTER
    Check: CHECK
    Check-letter: CH-LETTER
    Chem/physics: BOOLEAN
    Constituents:
        Answer: letter.Reply.Decision
        Boards: CB-SCORE (multiple)
        Forms:
            Evaluation-forms: RECOMMENDATIONS (multiple)
            Final-application: FINAL-APPLICATION-FORM
            Secondary-school-report: SCHOOL-REPORT
        Interview-report: INTERVIEW-REPORT
        Preliminary: PRELIMINARY-APPLICATION-CARD
        S-S-G-R: SECONDARY-SCHOOL-GRADE-REPORT
        Transcript: TRANSCRIPT
    Decision: DECISION
    Early-decision: BOOLEAN
    Faculty-review: INTEGER
    Faculty-reviewer: REVIEWER
    Foreign: BOOLEAN
    Interviewer: Interview-report.Interviewer
    Letter: LETTER
    L-list: LAUNDRY-LIST
    Minority: BOOLEAN
    MITID: Student.Id#
    Name: Refers.Name
    Refers: APPLICANT
    Scholastic-Index:
        S-I1: NUMBER
        S-I2: NUMBER
    School: Student.High-school
    Staff-reviewer1: ADMISSIONS-STAFF
    Staff-reviewer2: STAFF
    Staff-review1: ADMISSIONS-STAFF
    Staff-review2: INTEGER
    Student: Refers
    Year-applying-for: YEAR
```

```
Procedure Undergraduate-admissions
Object: APPLICATION
Responsible: DIRECTOR
Main-line:

    null (event 1) → step 1 → waiting
    waiting (event 2) → step 2 → complete
    complete (event 3) → step 3 → reviewed
    reviewed (event 4) → step 4 → {admitted,reviewed,done}
    reviewed (event 8) → step 8 → {reviewed,done}
    admitted (event 5) → step 5 → accepted
    accepted (event 6) → step 6 → coming
    coming (event 7) → step 7 → done

    Events:
        1. Receive Preliminary
        2. Receive Final-application-forms
        3. January 20, Year-applying-for
        4. February 20, Year-applying-for
        5. Receive Reply
        6. July 15, Year-applying-for
        7. September 30, Year-applying-for
        8. April 15, Year-applying-for

    Steps
        1.1 Verify Preliminary
        1.2 Select Interviewer from INTERVIEWERS
        1.3a Send Final-application-forms
        1.3b Send Interview-report to Interviewer

        2.1 Verify Forms.Final-Application.Check
        2.2 Send Check to MIT-OFFICES(Name="Cashier")

        3.1 Send S-S-G-R,L-list
        3.2a AA Select Faculty-reviewer from FACULTY
        3.2b AA Select Staff-reviewer from ADMISSIONS-STAFF
        3.3a Faculty-reviewer Determine Faculty-review
        3.3b Staff-reviewer1 Determine Staff-review1
        where absolute value (Faculty-review - Staff-review1) > 1 add
            3.4 AA Select Staff-reviewer2 from ADMISSIONS-STAFF
            3.5 Staff-reviewer2 Determine Staff-review2
            end
        where CB-scores = "unknown" add
            3.6 Send CB-letter
            end

        4.1 Calculate Scholastic-Index
        4.2 Group APPLICATIONS into ADMITTEDS, REJECTEDS, WAIT-LISTEDS
        4.3 Create Letter
        4.4 Send Letter
        where Decision = "admit" end in admitted
        where Decision = "reject" end in done
        where Decision = "waitlist" end in reviewed

        5.1 Create ADMITTED.Acknowledgment
        5.2 If Acceptance = "refuse" add
            5.3 Send E3 to Financial-Aid
            end in done
        5.4 Send AAC to Financial-Aid

        6.1 Send each ADMITTED(copy) to MIT-OFFICES(Name="F-A-C")
```

```
        7.1 Send E3 to MIT-OFFICES(Name="F-A-C")
        7.2 Archive each APPLICATION

        8.1 Group WAIT-LISTED into {ACCEPTED, REJECTED}
        8.2 Create Letter
        8.3 Send Letter
        where Decision = "reject" end in done
        where Decision = "admit" end in admitted

Quantitative Information:
    Timing constraints:
        1. Event 2 < Event 4

    Procedure statistics:
        Objects: 4500
        Variations:
            1.  300
        Branching:
             Step 4 → admitted: 44%
             Step 5 → accepted: 50%

Exceptions:
    Timing constraint:
        1. Send LATE-LETTER
    Activity-specific:
        1.1 Unable to verify:
            1.1 Send Preliminary,Problem-letter to Student
            1.2 Terminate
        2.1 Unable to verify:
            1.1 Send Check-letter
    General:
        Cancellation:
            where > Event 4
                1.1 Send CANCEL-ACKNOWLEDGMENT
                1.2 Terminate

Variations:
    1. where Early-decision = "T":
        add:
            reviewed (event 9) → step 9 → {reviewed, admitted}
            Events:
                9.  November 30, Year-applying-for - 1
            Steps:
                9.1 Calculate Scholastic-Index
                9.2 Group APPLICATIONS into ADMITTED, DISCOURAGE
                9.3 Create Letter
                9.4 Send Letter
                where Decision = "admit" end in admitted
        replace:
            Events:
                3.  November 20, Year-applying-for - 1
```

```
2. where Foreign = "T":
    add:
        wait-FSO-ok (event 10) step 1
        Events:
            10. Receive Preliminary.FSO-reply
        Steps:
            5.1-5.4 same
            5.5 Set Mit-Id
    replace:
        null (Event1) → step 10 → wait-FSO-ok
        Events:
            3. January 31, Year-applying-for
        Steps:
            10.1 Send Preliminary to MIT-OFFICES(Name="FSO")
```

While independent of OAM, the concepts of OSL provide the basis upon which OAM has been designed, and OSL supports and guides the OAM process. On the other hand, the perspective on office work on which OSL and OAM are based is independent of the particular linguistic constructs of OSL. Indeed, the English language writeup that OAM produces is structured along the lines of an OSL description. In order to emphasize these underlying concepts, we are developing several forms of *skeleton* OSL descriptions. An OSL skeleton is a formally-structured English description of the office. It uses the framework and templates of the OSL function and procedure models, as well as the built-in types for identifying important entities in the environment. The operational skeleton consists of filled-in function template(s) (naming the resource, the procedures, and the relevant events) and abstracted procedure descriptions. A procedure abstract uses the OSL procedure template form, including the main-line state diagram; however, a step is specified not in terms of OSL activities, but simply as a short declarative English sentence. Similarly, events are described in short phrases rather than by means of OSL syntax and semantics.

In other words, an OSL skeleton eschews much of the formality of OSL language while retaining its basic structure. An OSL skeleton is a canonical framework for organizing an office description, and is closely related to the English-language write-up produced by an OAM analyst. In our current efforts to further integrate OAM and OSL, we are incorporating the production of an OSL skeleton into the OAM analysis process.

OSL provides an office-oriented vocabulary of verbs and noun classes with which the analyst thinks about and describes the office. The formal framework of OSL assists the analyst in organizing his interview, checking for missing pieces and inconsistencies, and communicating his findings to his coworkers for comment and correction. Users of OSL have found these capabilities to be among valuable contributions to their practice.

Besides making incremental enhancements to OSL that respond to problems encountered in actual use, we have been working on two major extensions to the language. The first involves analyzing the decision activities that are part of the primitive vocabulary of the language, in an attempt to provide more insight into the analysis, description and support of these critical procedure components. The second is the extension of the language and analysis methodology to more readily support the reorganization of office operations.

In the first instance, we want an OSL procedure to include more information about how its constituent decision activities are performed. Such information might include the data upon which the decision is based, what factors constrain it, and the criteria that determine the quality of the decision. This information contributes to the

precision of the OSL description, and is particularly valuable for office rationalization and automated system design. While such analysis has something in common with techniques employed in the development of "decision support systems" (DSS), our context is somewhat different than the typical environment for which a DSS is built. Office decisions are more frequent, somewhat more structured, and more operationally oriented than those on which DSS research has focused.

We have begun a study of the information requirements of each of the OSL decision activities, and are extending the language to include this additional dimension. Our research methodology is based on the examination of a large number of examples to identify and abstract common structures.

The second extension to OSL is intended to enhance its usefulness as a tool by which procedures and functions can be rationalized or reorganized to more effectively achieve the overriding mission of the office. To this end, we are expanding OSL to capture the "intention" of a procedure; that is, an OSL description should provide information as to *why* the procedure is being performed. Our goal is to understand the various levels at which the purpose of a procedure can be described, and to develop a taxonomy of the most common purposes at each such level. This taxonomy will drive the selection of OSL language constructs to enable this information to be precisely and naturally expressed in a procedure description.

We have identified three levels of abstraction for describing a procedure's intention, which correspond to different aspects of an OSL specification. The first is at the activity level, which specifies the individual actions performed to realize the procedure. The questions that should be asked (and answered at this level include: Why is this activity being done? Does it really need to be performed? if so, how could it be accomplished more effectively in the context of the given procedure?

The second level of intention is that of the procedure itself. Relevant issues here include: Why is this procedure being performed? How does it interact with the other procedures that constitute the function? How does it help achieve the office's mission?

The third level of intention is at the office or function level. Why is this function spread out over several offices? Why is the organizational chart structured as it is? What goals of the firm does this organization serve? At this level, techniques and experiences in the areas of organizational development and design become important.

The goal of these investigations, is to improve OSL's abilities to describe and communicate a set of goals, requirements and implementations. Advances in OSL's

expressive abilities will demand extensions to OSL's analysis techniques, to enable an analyst to elicit the additional information. However, we expect that, as in the past, advances in OSL will provide the conceptual framework for OAM and consequently guide its evolution.

OAM and OSL are tools that embody our perspective on office automation. In order to evaluate the utility of these tools, their ease of learning, and the validity of our overall approach we have undertaken a series of field studies with a number of cooperating firms. In these studies we have presented a three-day training seminar on OAM and OS to personnel from the participating companies. These personnel have then returned to their organizations to apply our techniques to operational offices there, and to provide us with feedback on their experiences. The participating organizations have evinced a willingness to participate in our research project, with the risks that such an effort entails. The individuals involved have agreed to try our approach and techniques, both to experiment with new tools that may assist them in doing their jobs better, and to explore the benefits that may accrue from presenting their specific problems to us as case studies upon which further research will be based.

Approximately 40 people, representing seven firms, have attended courses designed to present both an appreciation of our approach to office automation, and OSL and OAM as means of implementing that approach. The firms include several insurance companies, a research laboratory, a chemical manufacturer, a consumer products company, and an industrial products manufacturer. The attending personnel involved have had a wide variety of training and backgrounds, ranging from data processing systems analysis to industrial engineering, line management, secretarial, and shop foreman experience. Two courses have been given this year, in November at MIT, and in January in Rochester, NY. (Another half-dozen participating firms are awaiting scheduling of a course in New York, which will be a revised version based upon results from the present studies.) About twelve studies are currently underway in several types of offices. We have already received feedback, including initial OSL writeups, on about half of these.

These courses and studies have been exceptionally fruitful, even in their early stages, in helping us to understand and enhance our material. While some inadequacies bugs, (not unexpectedly), have appeared, we have been gratified by the response of those undertaking studies. Most participants have found our approach sensible and easy to understand. OAM has been received well, particularly by those with little or no experience in office or systems analysis. OSL is more difficult to teach and learn; as with any formal language, use is the best teacher. The reaction of those who have attempted to use it, however, has been quite positive, particularly in the way that it structures the analysis process and organizes the documentation at various stages of a project. In fact, several users have used

several levels of OSL skeletons as replacements for, rather than in addition to, the draft English writeups recommended by OAM.

Preliminary results from the courses and field tests have shown us ways to improve our methods of teaching the material, as well as identifying problems with, and limitations of, the tools themselves. We also expect these collaborative efforts to contribute to our research in other ways. Most of the activities that we have underway or planned in the area of office analysis and office system requirements analysis (described below), have been prompted or affected by the field studies. These studies are also assisting us in building up a database of office descriptions for further research. Finally, we are beginning to follow and to assist in, a number of automation projects that have begun with the use of OAM and OSL, to see how they can be used to guide the design and implementation phases of an office automation effort.

3.3. Office Productivity Analysis

John Bakopoulos has begun a study of office productivity measurement.

Using *economic* value, as opposed to book or *financial* value, can be used as the basis of a scheme for productivity accounting, by creating *internal markets* for goods and services provided and/or used within an organization. This scheme will allow the *value added* at each profit center to be determined (for measurement purposes) and will introduce incentives to cut down overhead costs for both the supplier and the user of the associated goods and services.

A transfer pricing scheme usually implies a decentralized, profit-center oriented organizational structure. Its main objectives are to provide valid data for make-or-buy, pricing, and capital budgeting decisions, which should be made at the most appropriate (best informed) level, as well as to provide data for divisional evaluation and control. On the other hand, problems of goal congruence, incentive and autonomy become very imposing in many situations, as does the determination of the appropriate transfer prices. It is our belief that most of these problems can be solved via internal regulation of the markets involved, an area that we are currently investigating.

A quality (control) circle is a small group of people who perform similar work and meet voluntarily on a regular basis to identify, analyze and seek solutions to work-related problems. This technique has been extensively used in Japan, principally in manufacturing environments, and reportedly with great success.

The circles operate in small groups (4-10 people) usually led by an appropriately trained individual; everyone willing to participate is welcome to. During quality circle

meetings, problem areas are identified, relevant causes are investigated, and a solution is determined; after the solution is sold to management during a subsequent presentation, it is put into practice.

Quality circles provide significant advantages, including the exploitation of the know-how and creativity of employees, the effectiveness of small group meetings combined with brainstorming sessions, and significant enhancement of job quality and employee satisfaction. They work best in environments where effectiveness is more important than efficiency.

Introducing the quality circle idea into the office seems to be relatively easy. Using them for office automation, however, can be a problem, because of the lack of technical expertise among most employees. One possible solution, is to have a technically-oriented analyst *lead* the appropriate circle meetings; however, this is not straightforward if the original nature of the circles is to be maintained.

Traditional definitions of productivity in terms of inputs and outputs fail to capture many important aspects of the office environment. An alternative approach is to ask the people themselves to identify what really matters in their work, what factors should be taken into account for evaluation purposes, and what office products and processes deserve attention. This process seeks to identify the *critical success factors* of the office.

Along with the notion of critical success factors comes the dual notion of *major inhibiting factors*, those aspects of current office operation that impede its successful operation. An analysis of these major factors can lead to the identification of the key aspects of the office whose automation or support will yield the greatest benefits.

In order to further develop these notions, and to assess their utility, we are planning to undertake a number of experimental pilot projects in the coming year, both with MIT offices and with other organizations, in which we will seek to employ these approaches in the context of actual automation efforts.

3.4. Plans

In the coming year, we intend to continue our research in all of the above topics: office analysis, office specification, office productivity measurement. Moreover, we plan to extend our activities to the full range of issues entailed in the implementation of functional office automation systems. In particular, we plan to develop a complete office *automation* methodology. In particular, this methodology will provide guidance for the of office procedures and for the design and implementation of automated systems.

In order for this methodology to address the substance of an office's rather than the artifacts of its operation, it will be based on a taxonomy of the kinds of *functions* that are typically performed in offices. Based on our studies of a large number of offices, we have discovered that seemingly different and unrelated offices can often be viewed as instances of a more general and abstract office types. For example, a doctor's office, a television station, and a flying school can all be usefully seen as *schedule-oriented* offices. Moreover, the same kind of automated system, suitably tailored and customized, could be successfully employed in each of them, thereby obviating the need for constructing from scratch three independent and unrelated systems.

We are currently engaged in an effort to identify as complete a set as possible of these office functions. As usual, our methodology is based on the study of a wide variety of operational offices in different organizations. For each *function*, we also plan to identify the key *applications* performed in realizing it and the major automated *tools* useful for these applications. Our goal is to provide an analyst with a tool that will enable him to assess an office's automation requirements. By utilizing our taxonomy, the analyst should be able to relatively easily identify the principal functions and applications of the office and then to translate this into an appropriate system specification. Moreover, we plan to develop a set of function-specific automation guidelines, which will identify key issues, measures, techniques, and problems that will have to be addressed in automating the function in question. In this way, we hope to streamline and reduce the cost of the development of a custom, functional office support system.

4. MULTI-PERSON INFORMATION WORK

4.1. Introduction

Our research in Support Tools for Multi-person Activities is aimed at identifying protocols that people use when they are working together and investigating ways in which they could be better supported by additional software tools. People working together may rely on any of a number of communication protocols varying from complex patterns of message passing, to conventions for sharing a visual space (e.g., taking turns at the blackboard), to procedures for controlling order of verbal presentations (e.g., taking and yielding "the floor" at a meeting). Different kinds of joint work make use of particular protocols in varying degrees. For example, a group trying to schedule a conference may rely most heavily on an exchange of messages, by mail or by phone, to gather scheduling information, negotiate a time, reserve tentative times and confirm or revise decisions. By contrast co-authors of a paper might both have access to the draft and could simply take turns revising the

document. If only one author has authority to make changes to the paper, the other might send his comments on a draft to the first author, thus using the document itself as a message medium.

The typical office workstation or personal computer provides support for a number of individual activities (text processing, data base queries, various special purpose applications packages, etc.) as well as support for one kind of communication, namely mail or messages. However, in many kinds of close working relationships, mail is a rather indirect and awkward communication mechanism. A text editing system could support joint authorship by bringing one author's changes to the attention of the other at the appropriate time (e.g. when the first author indicates that he has completed a set of revisions.) This saves the author from having to compose and send mail saying, "look at the change I've made to our paper in file XYZ.TXT.55." A real-time communication support system, could provide a "shared visual work space." A group of engineers trying to design a circuit together might work at a distance using this shared space. Special function keys could allow one to request a turn to write in the space and users could be prevented from writing at the same time in the same part of the screen. By contrast, simple terminal linking leaves it to the workers to agree to and enforce ad hoc protocols for synchronizing their use of the screen -- the system simply echoes characters typed by any linked user on all linked terminals.

Our conjecture is that users can be better supported in joint work by a system designed to meet their application-specific communication needs than they are by a system providing application support along with separate mail and message support. We are developing design principles and software tools that can be used by the application designer and programmer when building systems to support joint work. Thus rather than trying to provide general purpose communications tools to the end-user directly, we expect to find generality in our work at the lower level of software tools for building new systems.

4.2. Progress

This year we have been designing and implementing a prototype system (the calendar), conducting studies of a series of contrasting applications, studying general meeting activities, and implementing some real-time communication protocols. The calendar design has led to identification of a number of kinds of person-to-person protocols and means for supporting them. It will be our first source of user feedback on our approach to providing application specific support for multi-person activity and will provide a preliminary framework within which to analyze other applications. The real-time support package development has raised a number of interesting problems which will be the subject of the doctoral research of Sunil Sarin.

4.3. The Calendar

The design of the calendar has been based on interviews, group discussion, and observation of our current practices using both electronic mail and paper calendars for scheduling group working meetings. One reason for actually building the system is that we do not as yet know how well we can judge the needs of a user from their descriptions of the current functionality of existing support systems. Studying existing meetings or computer conferences may prejudice us towards supporting protocols that are based on current media -- face-to-face or computer mail, paper or electronic messages -- and not allow us to see some of the effects of building communications support into the application system. Implementation and use of a prototype system can help us to test out some of our ideas quickly. To this end we chose a simple application area and have been building as many different communications support tools for it as we can imagine (or as are suggested by potential users). The calendar system is *not* a system that is very clever about scheduling. Rather it provides appropriate communications and decision support to allow users to conveniently make scheduling decisions themselves.

The fall term was dedicated to building up some functionality for personal calendars before adding communications facilities. The result was a personal calendar in which one could make several kinds of appointments, cancel or reschedule or otherwise modify existing appointments, display appointments for a day or week, add reminders and look at other people's calendars. There are special kinds of appointments for calling meetings and arranging seminars. If you are arranging a "meeting" you can specify the participants. If you arrange a "seminar" you can specify a variety of items relevant to seminars including time for refreshments, speaker, host, etc.

This summer, a new version of the calendar will be available on the Dec 20. This version will support meeting scheduling and calendar sharing. By September several "fancier" communications facilities such as more flexible scheduling of tentative meetings will be available. We expect people in the Office Automation Group to use the calendar for personal as well as group scheduling this summer (some have already experimented with earlier versions to provide us with user feedback.)

The aspects that we have been concentrating on for the summer version are;

1) Data base issues -- making the data base reliable and sharable.

2) Communication through shared data -- implementing an access rights scheme so that an owner can specify which people can share his calendar and what operations they can perform. Certain stylized sets of

access rights called "roles" will be supported so that a person can be given, say, "secretary" or "owner" rights to the calendar.

3) Communications through message passing -- Adding a message passing facility that will allow people to send requests to other calendars. The meeting command will automatically use this feature to coordinate with and confirm appointments with other people. Other features that will support this function include facilities for examining other people's calendars, allowing people to provide public scheduling information, facilitating merging of schedules so that the caller of a meeting can look for likely meeting times. Also, it will be possible for a user to find out if there are new appointments.

4) Personal Calendar Features -- Adding certain personal calendar features such as the ability to schedule regularly occurring appointments over a certain period of time and refining certain existing features so that, for instance, users will be able to more conveniently edit a day's entry (to eliminate reminders, change reminders, etc.)

5) Interface --Redesigning the user interface so that the state of the user's interaction with the system is remembered. This includes such information as a current day that can be used as a default if the user does not wish to type the date. The screen organization and command entry have been redesigned to include improved command parsing. Long series of prompts have been replaced by pop up forms that can be filled in using a forms editor.

Communication through Shared Data: We have implemented the "roles" of secretary and owner this term. Typical protocols based on these roles might be the following:

- If a secretary puts an appointment on his manager's calendar due to a telephone request from someone outside the system, the manager should see that appointment flagged so that it is brought to his attention.

- If the manager doesn't want to keep that appointment, he would like to delete it from his calendar, but still know that on his secretary's version that appointment shows up with an annotation indicating that a call should be made to cancel or perhaps to reschedule.

The current design calls for addition of an access file in which user names can be associated with roles, functions for changing settings in this file, and modification of calendar functions so hat they are executed in accordance with the role of the

current user. Display features such as flagging appointment listings will be needed to bring to the manager/secretary's attention changes that may require attention. Some communication will also take place through the comments fields in the appointment form itself. While most communication is through shared data, the the user interface for processing new items will resemble that provided for Message Passing.

This part of the calendar will be the subject of a bachelor's thesis by Fred Barrett.

Message Passing: The meeting capability being implemented is a precursor of the full meeting scheduling protocol that will eventually be part of the calendar (*cf.* working paper 025). In this version message passing can be used to request an appointment with another person. The appointment can be made on one's own calendar but will be flagged as having a "request outstanding." If and when a positive response is received the flag will be removed. The message passing mechanism will eliminate the need for explicit composition and transmission of messages through mail when in fact the content is based on calendar actions.

To support the user in this form of communication there must be operations for examining new requests, answering requests, and processing answers. It must be possible to associate answers with the requests to which they refer. When changes are made to an appointment that is known to appear on other calendars, messages will have to be sent out to inform others.

Interface : The screen will contain a status window, command line window, and a main display plus some pop-up windows for error messages and other notices. The status window at the top of the screen (one line) displays the following information: name of the calendar, current day, current appointment and an indication of whether there are new requests.

The command line window will be at the bottom of the screen. The rest of the screen will generally contain the main display of calendar information. Sometimes a part of the main window will be covered by a pop up form to be filled in by the user. Generally, the user will be able to fill in an appointment form while still seeing the relevant day's appointments listed in the visible part of the main window. Occasionally a "caution" window will appear over the upper right hand corner of the main window to display a message relevant to the operation (such as a warning that scheduling changes being made to a form during editing are subject to availability check). This area will also be used for menus of commands when requested by the user. A message window can pop up above the form or command window if there is an error or exceptional condition to be brought to the user's attention. This window will be used for confirmations as well -- the user will be able to type in responses such as c (commit), a (abort) or r (revise).

The command language will follow some principles established in the ETUDE/ECOLE design. Commands will be designed to be verb-object phrases, but for some commands the object alone will determine the verb. The verbs will be **list** (a day), **show** (a day), **make** (kind of reservation), **cancel** (appointment id), **examine** (appointment id), **choose** (appointment id), **change** (calendar id), **remove** (appointment or note id), **add** (appointment or note id), **answer** (appointment or message id). A command line consisting of a date only, will be interpreted as a request to **list** (or **show**) that day, changing the current day value. A command line containing date and time specification will be interpreted as **make** that appointment. If an appointment is made on a date other than the current day, the date of the appointment determines the display temporarily. After confirmation this day is remembered as "last" and the display returns to the current day. A command to "**list** last" changes the value of current day to "last."

The forms editor will provide a uniform user interface for filling in forms. It will be possible to move from field to field, modify or erase fields, enter information to calendar, abort an editing operation, move from the form to the command window, etc. The forms editor is invoked by calendar operations by sending an object of type reservation plus information in table form about fields that **must** have values, fields that **may** have values, types of fields (or names of parsing routines and possibly consistency constraints among fields (by specifying names by consistency checking routines).

Two undergraduates have been implementing the interface. One of them, John Wenn, is writing a bachelor's thesis on the design of the forms editer.

4.4. Real-time Communication

While "delayed mode" communication, in which users interact with a database or send messages to each other asynchronously at their own convenience, may be adequate for many stages of problem-solving activity, the need to negotiate and thrash out details in the final stages frequently requires simultaneous interaction. To address this need, we are developing methods to support *real-time sessions* in which users conduct a "meeting" by computer. Such a session would typically (but not necessarily) be augmented by a voice and perhaps even a video link, to enable free discussion of the problem at hand without artificial constraints imposed by the system. (While we consider auxiliary communication media, especially voice, to be important in this regard, they are not the main focus of our research.) The primary role of the computer system, then, is not to transmit text messages between users (which is what "computer conferencing" systems provide, and which we will still retain as an option), but rather to enable manipulation of on-line problem data in such a way that it is simultaneously visible to every participant in the session.

The kinds of facilities that could be provided in such a real-time session are best illustrated by an example software system that Sunil Sarin has developed to support meeting scheduling. A user who wishes to schedule a meeting takes on the role of session *chairperson*; he types in the names of the desired meeting participants, the range of acceptable dates for the meeting, the desired meeting duration, and a short description of the purpose of the meeting. A "session invitation" message is then sent to those participants who are currently on line and running their calendar programs. (These participants could have been contacted by phone, and instructed to start their programs, just prior to starting the session, or a phone link could be established after initial rendezvous has been made over the system with those participants that happen to be available. The system is sufficiently self-contained that the session could be held even without any phone connection, although that would preclude any voice discussion.) A participant receiving this invitation is asked by his program whether or not he wishes to join the session. If he agrees, a message is returned to the chairperson carrying an "outline" of the relevant range of dates extracted from the participant's personal calendar; this outline indicates which time slots are free and which are not, without disclosing any additional information such as why the participant might be unavailable at a given time. The participant may alternatively decline to join the scheduling session (which does not mean that he cannot come to the planned meeting, whenever it may be held), and a message to that effect is accordingly sent to the chairperson's program.

The real-time session begins in earnest when the chairperson's program receives replies from all participants who are on line (or when the chairperson instructs his program to stop waiting for replies and commence the session, which might be necessary if some replies are not received after a long wait). At this point, the calendar outlines received from the participants who joined the session are "merged" to identify time slots for which all participants are available. The displays of the session participants are initialized to show this merged view. This aggregation of information from the various participants' personal calendars constitutes the *shared information space* of the session. It can be manipulated by commands to "scroll" the displayed view over the range of dates, and by commands that select different combinations of participant calendars for merging (which is necessary in case there are scheduling conflicts that cannot otherwise be resolved); the effects of each such command are immediately displayed on every participant's terminal.

At any given time, only one of the session participants is allowed to enter commands to manipulate the displayed merge; this participant, initially the chairperson, is said to be *in control*. Special commands are provided to allow the participant currently in control to "give the floor" by passing control to another participant (or back to the chairperson). A "request control" command is also provided; the system queues all participant requests, and a list of requests that have not yet been granted can be displayed on command. The chairperson is given

special powers in that he can at any time *preempt* control from the participant who currently has it. These "session control commands" have no effect on the shared display; they only determine who has the ability to manipulate this display.

Each participant in a session sees not only the shared display but also a *private window*. Whereas the shared display shows a gross view of the participants' calendars indicating only whether or not a given time slot is free, the private window contains more detailed information, such as descriptions of or comments about this participant's appointments, that is truly private and is not visible to the other participants. The purpose of showing this private data is to aid decision-making - seeing details of his private appointments can help the participant decide whether or not he can agree to a proposed meeting time (see below). The private window is displayed side-by-side with the shared display and is always kept "aligned" - whenever the shared display is updated to show a different date and time interval, the private display is automatically updated to show the same interval. This ensures that the displayed private information is always current and relevant with respect to the shared display.

Decision-making in a session is supported by allowing participants to *vote* on a *proposal*. The participant currently in control can propose a specific date and time for the meeting, and each participant enters a Yes or No vote depending on whether or not he finds the proposed meeting time agreeable. (A participant is not constrained to base his vote on whether or not his personal calendar shows him as being available.) The votes are tabulated and displayed to all participants as they are made. At the end of the voting, the chairperson can *commit* the proposed time if he so desires, or he can continue the session (in which case alternative meeting times might then be proposed and voted on). If the chairperson does commit a final meeting time (which he can do at any point in the session, not just immediately following a vote), the participants are informed and their personal calendars updated to reflect the chosen meeting time; the session is then terminated. Alternatively, the chairperson can *abort* the session at any time, without any final meeting time having been chosen.

A participant can leave a session permanently at any time. He can also "escape" from the session temporarily to perform some unrelated activity at his terminal (such as read some newly-arrived mail). When he "returns" to the session (by resuming his session program), his display is brought up to date to reflect the current state of the session; he is also informed of any important events that may have occurred in his absence, such as votes on proposed times or the committal of a final meeting time.

4.5. Some Observations

Delayed Mode Communication: We have begun to focus on the relationship between communication through shared data and communication through message passing -- the distinction is fuzzy and we see clarification of the similarities and differences as an important aspect of our work. For example, when one changes a shared data base and wants that change brought to someone else's attention the data base change may in effect cause a message to be sent. In fact, our shared data communication support (*cf.* the calendar description above) provides support that is quite similar to that of the message passing facility. The original design notes emphasized the distinction between shared data and message passing based on the kinds of communication that we saw happening with real calendars -- people with access to the same hard copy (shared data) don't seem to have to do quite the same kind of communicating to set up appointments that is required for coordinating among people who cannot see each other's calendars. Protocol design may include conventions about frequency at which one will be accessing shared data -- if the data base may not be examined regularly then a change to the data base may have to cause a message to be sent as well.

We have been considering when it is appropriate in the implementation to use a shared data base protocol as opposed to a message passing one. The same decision has to be made for the user interface which may emphasize one view or the other independent of implementation. A major consideration in design will be the "naturalness" of one view -- when does the user want to think in terms of messages and when in terms of sharing data.

Given that there will be some message passing in delayed mode communication, the activity structure and interaction mechanisms must support smooth transition from application specific operations (in the calendar these are operations dealing with appointments) to message passing and back. Thus messages should be composed automatically when their content is obvious from the calendar operation (sending an invitation to a participant in a meeting). Also, calendar operations should be executed easily (perhaps even automatically) as the result of message passing operations (e.g. a positive reply to an invitation ought to cause the invitation to appear on the invitee's calendar). Messages ought to be easily connected with relevant appointments so that data relevant to composing an answer can easily be accessed.

Real-time Communication: In real-time communication we have identified a variety of useful mechanisms for joining or leaving a session, voting, asking for control of the shared screen, etc. There is also need for coordination between and separation among kinds of data. Shared data will often incorporate a filtered (public) version of each individual's private data and individuals will often want to keep both versions visible and synchronized, as in the prototype system described above.

It is not clear that there is a single most-appropriate way to implement the shared space used in a real-time session.

- Simple linking of terminals is not generally adequate -- there should be separation of the writings of the various individuals. Also, individuals in the session should have the ability to do things that are independent of the session (perhaps only in other windows).

- One approach is to dedicate only a part of each screen to the shared data. The question then is how to get the data to each user. Each participant can send his data to the "chairman" of the session, the chairman can form the shared display from that data and distribute the information to the session participants. Alternatively the participants can broadcast information to other participants and each participant can form the display from that data.

- The advantage of the first distribution mechanism is that there is one place to which changes have to be sent, and it is easy to keep all displays identical based on the current information. The advantage to the second would be that each user would actually have available the data from which the display was formed. A place where this would matter is in transferring information between public and private windows. Say the user wants to take the current public view into a private work space, manipulate it without effecting the data base or the public view. If he only has available the character strings displayed at his terminal, he may be restricted in the operations he can perform. (E.g., in the calendar where the public display is a merged version of the filtered calendars of the participants, he may not be able to remove some individuals from the set to see how that affects the group availability.)

- Finally, if the data seen in the shared space is made up of data from various private data bases then termination of the session can be complicated. The shared data may have been modified on the screen as part of the work of the session. If finally the consensus of the group is that the data in the shared space should be recorded permanently in the actual data bases, there can be considerable work left in updating consistently all of the involved private data bases.

4.6. Software Packages

We have begun to characterize some of the software support that might be of use in building applications programs that support either real-time or delayed communication. The following outline covers some of the issues and desired functionality of these packages:

- To support real-time shared space we need mechanisms for:
    * passing control
    * echoing on terminals
    * composing the shared view
    * entering and leaving
- Most window packages would support definition and placement of windows. In addition, for real-time communication, we would need support for:
    * specifying relationships between private and public windows
    * ways to synchronize data in windows
    * a mechanism for linking a window with other a part of another person's screen (perhaps for private conversations within a session)
    * ways to transfer data from window to window (particularly interesting between public and private windows)
- The database may be designed to support communication by:
    * sharing
    * linking data changes and to communications
    * support for a distributed update to bring data into agreement with the final state of the shared workspace
- The message passing facility may interact with the mail system if there are mechanisms for:

* packaging up relevant information to compose a message

* specifying text formatting based on data type definition

4.7. Meeting Activities

There is some amount of literature on meetings and working groups that we are surveying to see what is already known about how people interact without telecommunications or data communications support. There is also already a fair amount of literature on teleconferencing and electronic meetings. Survey of this literature will enable us to direct interviews and studies of groups that are working together to find out what protocols they use to accomplish their work. A bachelor's thesis is being written on general meeting activities as reported in the social sciences literature.

We would also like to begin some studies of our own on meeting activities. This kind of information could be gathered from interviews, observations of working groups and meetings, case studies of the sort done in the OAM/OSL project, etc. We have been postponing this effort until we have more experience with the prototype system so that we will have a framework suggesting new kinds of things to look for.

4.8. Other Applications

We have considered a number of other applications that might contrast with the calendar. Most recently we have focused on developing a set of scenarios for joint document writing. We have been working with the ECOLE group to identify places where the current text processing or data base facilities would not adequately support these scenarios as well as places where features required for joint work would in fact be of benefit even to the individual user. Examples of the latter arise in various ways of managing versions, copies and comments.

4.9. Plans

The plans for next year include extensions to the calendar work, design of the joint document writing system, continued development of the real-time support packages, as well as definition of related research projects particularly in human factors and use of other communication media.

4.10. Calendar

Next year we will begin to explore the issues of making multi-person support tools available on a heterogeneous hardware base by working with a group of UROP students to make the calendar available on other machines in the building. We will also be adding new features to the calendar to make it more useful while expanding its communications facilities. We are designing a things-to-do and tickler file feature that can be used by the meeting protocol to keep the caller appraised of the status of unconfirmed meetings. We will also consider adding a variety of public calendar facilities. Public calendars can be used to augment personal calendars for instance to have seminars and public events appear as if they were in one's calendar without having to dedicate private calendar space to recording the information.

We also plan to evaluate the user reaction to the design and will be considering ways of gathering information and of making use of this feedback. User reactions should aid us both in planning modifications to the calendar system and in design of other application systems.

4.11. Real-time Communication

In the course of constructing our prototype system, we identified several architectural alternatives for implementing real-time sessions in a distributed environment. (Although our experimental meeting scheduling system is implemented on a time-shared mainframe, it does use interprocess message-passing to simulate distribution.) These alternatives represent different combinations of compatible answers to the following questions:

- How much of the shared database is replicated by having every participant's workstation (or program) store a local copy of it?

- How is the ability to manipulate the shared information controlled? (In some applications only the participant currently "in control" should have this ability; the participants' session programs must cooperate to ensure that this is the case, and respond correctly to the chairperson's and participants' commands to transfer control. In others we may need to provide more flexible access.)

- How is responsibility for processing commands (parsing of commands and arguments, followed by execution of the corresponding operation on the shared data) divided between the workstation of the participant issuing the command and the chairperson's workstation?

- How are the various participants' displays updated as a result of commands performed on the shared data?

Having identified these alternatives (and having implemented one of them in our prototype system), we next propose to study the conditions which might make one more attractive than another. The design choices will be constrained not only by the hardware and software implementation environment, but also by the nature of the desired user interface; the latter will often depend on the application for which real-time session support is being designed.

We also propose to examine ways of relaxing the constraint that only one participant at a time can perform operations in the shared space. We believe this constraint to be a useful one that users will want to follow most of the time. However, it might be too constraining at times, and users might wish to occasionally "diverge" from the main focus of a session or interact in the shared space concurrently while keeping the results of their interactions visible to all participants (perhaps by splitting the display screens into several windows). This will require devising both a command set for participants to specify their desire to diverge or resynchronize, and implementation-level techniques for processing the participants' inputs and manipulating their displays in accordance with the specified degree of asynchrony.

We also wish to pursue the integration of real-time and "delayed" communication facilities. In the context of meeting scheduling, this would allow a meeting caller to send "mail" style communications to participants who are not simultaneously on line (or who do not wish to join the real-time session); the participants can then process and respond to these communications at their own convenience. Similarly, users should be able to send calendar outlines in delayed mode so that a single user could perform merging operations to find suitable meeting times.

It has become clear that real-time sessions should be supported by auxiliary voice channels. We will have to provide a facility for telephone conferencing so that session participants can talk while in the session. Experimentation with the use of this loosely coupled voice channel when the calendar prototype is complete will help us to evaluate the need for fancier voice support in the future. We expect eventually to incorporate some automatic dialing facilities into the system so that sessions can be started more conveniently.

4.12. Other Applications/Activities

We will be continuing case studies in applications areas, considering alternative implementation strategies and ultimately producing additional software tools that can be placed on an office workstation.

We have already identified kinds of meeting activities that do not arise very naturally in the calendar application. We think that the exercise of doing at least a fairly detailed design of some contrasting applications will provide us will better understanding of how different the requirements might be for one of these other activities. Such support tools do not necessarily fall in the class of communication protocols. One example is support for exploring alternatives during problem solving. At certain times it may be appropriate to record the current state of the data base (or working data) so that it can be returned to and reexamined later. Such recording points might be recognized automatically by the system based on application specific information (e.g. every item a paragraph is completed, every time a major component is deleted). In addition, users should be able to specify that a state should be saved and they should be provided with means for backing up, comparing two alternatives etc. The system should provide support for remembering alternative paths from such a saved point and ways of identifying differences. This kind of support is probably similar to that provided in general meeting support by logs of activities. Certain actions at a meeting would always cause log entries, others might be recorded based on a specific entry operation. Users might refer to the log to recall reasons for previous decisions, to catch up on what has been happening since they left the room, etc.

4.13. Related Research

We would like to formulate several human factors studies. One area of interest is establishing and clarifying the differences between computer conferencing, mail systems, and the kind of integrated support system that we are building based on information about patterns of communication and differences of usage.

Publications

1. Good, M. "Etude and the Folklore of User Interface Design,"

   Proceedings of ACM SIGPLAN/SIGOA Symposium on Text ManipulationPortland, OR, June, 1981.
   Office Automation Group Memorandum OAM-030, MIT Laboratory for Computer Science, Cambridge, MA, March 1981.)

2. Greif, I. "Notes on the Design of a Calendar System," Proceedings of the IEEE Computer Networking Symposium Gaithersburg, MD, December 1980.

3. Greif, I. and Hammer, M. "Multi-person Informational Work," Office Automation Group Working Paper WP-026, MIT Laboratory for Computer Science, Cambridge, MA, August 1980.

4. Greif, I. "Calendar Project: Progress and Plans," Office Automation Group Working Paper WP-029, MIT Laboratory for Computer Science, Cambridge, MA, March 1981.

5. Hammer, M., Sirbu M., Kunin, J. and Schoichet, S. "OSL/OAM Training Course," MIT Laboratory for Computer Science, Cambridge, MA, November 1980.

6. Hammer, M. and Kunin, J. "OSL: An Office Specification Language/Language Description," Office Automation Group Memorandum OAM-024, MIT Laboratory for Computer Science, Cambridge, MA, December 1980.

7. Hammer, M. and Zdonik, S. "Knowledge-based Query Processing," Very Large Databases Conference, Montreal, Canada, October 1980.

8. Hammer, M. et al. "Etude: An Integrated Document Processing System",

   Proceedings of the 1981 Office Automation Conference, Houston, TX, March 1981;
   Office Automation Group Memorandum OAM-028, MIT Laboratory for Computer Science, Cambridge, MA, February 1981.

9. Hammer, M., et al. "OAM: An Office Analysis Methodology", Office

Automation Group Memorandum OAM-016, MIT Laboratory for Computer Science, Cambridge, MA, January 1981.

10. Hammer, M., et al. "The Implementation of Etude, An Integrated and Interactive Document Production System",

    Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, OR, June 1981;
    Office Automation Group Memorandum OAM-026, MIT Laboratory for Computer Science Cambridge, MA, December 1980.)

11. Ilson, R. "Recent Research in Text Processing,"

    Words, June 1980; IEEE Transactions on Professional Communications, December 1981;
    Office Automation Group Memorandum OAM-031, MIT Laboratory for Computer Science, Cambridge, MA, March 1981.)

12. Ilson, R. and Good, M. "Etude: An Interactive Editor and Formatter," Proceedings of the 1981 Office Automation Conference, Houston, TX, March 1981.

13. Ilson, R. and Schoichet, S. "An Integrated Model for Formatted Documents," Office Automation Group Memorandum OAM-019, MIT Laboratory for Computer Science, Cambridge, MA, September 1980.

14. Ilson, R. and Schoichet, S. "An Integrated Model for Formatted Documents," Proceedings of the 1980 Computer Networking Symposium, IEEE-CS TC on Computer Communications and the NBS Institute for Computer Sciences and Technology, Gaithersburg MD, December 1980.

15. Kunin, J., Greif, I., Hammer M., Sutherland, J. and Sirbu, M. "OAM/OSL: Training Course," Rochester, NY, January 1981.

16. Kunin, J. "OSL: An Office Specification Language/Version I Reference Manual," Office Automation Group Memorandum OAM-21, MIT Laboratory for Computer Science, Cambridge, MA, February 1981.

17. Schoichet, S. "Case Studies of Office Procedures: The Office of Sponsored Research," Office Automation Group Memorandum

OAM-022, MIT Laboratory for Computer Science, Cambridge, MA, October 1980.

18. Schoichet, S. "Page Layout: Representation and Specification,"

    Proceedings of the International Conference on Research and Trends in Document Preparation Systems, Lausanne, Switzerland, February 1981.

    Office Automation Group Memorandum OAM-027, MIT Laboratory for Computer Science, Cambridge, MA, January 1981.

19. Schoichet, S. Review of The Office of the Future, Uhlig, R., Farber,D. and J. Bair (Editors), North-Holland Publishing Co., 1980. Computer Decisions, 13, 3, (March 1981).

20. Schoichet, S. "Personal Workstations: A Concept Evolves into an Industry," Mini-Micro Systems, 14, 4, (April 1981).

21. Sirbu, M. "Programming Organizational Design," Proceedings the Fifth International Computer Communication Conference, Atlanta, GA, October 1980.

22. Zarmer, C. and Kunin, J. "Case Studies of Office Procedures: The Industrial Liaison Office," Office Automation Group Memorandum OAM-020, MIT Laboratory for Computer Science, Cambridge, MA, October 1980.

23. Zarmer, C. and Schoichet, S. "Case Studies of Office Procedures: The Work Control Center," Office Automation Group Memorandum OAM-013, MIT Laboratory for Computer Science, Cambridge, MA, July 1980.

Talks

1. Good, M. "Etude and the Folklore of User Interface Design," ACM SIGPLAN/ SIGOA Symposium on Text Manipulation, Portland, OR, June 1981.

2. Hammer, M. "A Methodology for Office Systems Analysis,"

   Willis, Faber and Dumas, Ltd., London, England, January 1981;

Exxon Office Systems, New York, NY, April 1981.

3. Hammer, M. "A View of the Future,"

   Exxon Office Systems, New York, NY, March 1981;
   Management Assistance Inc, Los Angeles, CA, March 1981;
   Willis, Faber and Dumas, Ltd., London, England, January 1981.

4. Hammer, M. "Careers in Office Automation," MIT Career Development Seminar Series, Cambridge, MA, March 1981.

5. Hammer, M. "Design Issues in Integrated Office Systems," Digital Equipment Corporation, Merrimack, NH, May 1981.

6. Hammer, M. "Directions in Database Management," Willis, Faber and Dumas, Ltd., London, England, January 1981.

7. Hammer, M. "Doing Office Automation,"

   Life Office Management Association, Boston, MA, June 1980;
   Center for Information Systems Research, MIT Sloan School of Management, Cambridge, MA, June 1980.

8. Hammer, M. "Etude: An Interactive Text Editor and Formatter," University of Toronto, Office 80 Conference, Toronto, Ontario, September 1980.

9. Hammer, M. "Federated Databases," GTE Laboratories, Waltham, MA, July 1980.

10. Hammer, M. "Look into the Future," Willis, Faber and Dumas, Ltd., London, England, January 1981.

11. Hammer, M. "Office as System: An Approach to Office Automation,"

    Equitable Life Assurances, New York, NY, April 1981;
    Bell Laboratories, Holmdel, NJ, April 1981;
    Digital Equipment Corporation, Merrimack, NH, May 1981.
    Research Council Symposium on Office Automation, Washington, DC, July 1980.
    American Telephone and Telegraph, Basking Ridge, NJ, March 1981.

12. Hammer, M. "Office Automation and Electronic Mail," Postal Rate Commission, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

13. Hammer, M. "On the Design of Integrated Office Workstations,"Conference on Office Workstations, Andover, MA, September 1980.

14. Hammer, M. "Protestations on Productivity,"Wang User's Group Conference, Boston, MA, November 1980.

15. Hammer, M. "Research at LCS in Office Automation," MIT Industrial Liaison Program Symposium, Cambridge, MA, May 1981.

16. Hammer, M. "Software Issues in Office Automation,"

    Xerox, Office Products Division, Dallas, TX, July 1980;
    Gillette Corporation, Boston, MA, August 1980.

17. Hammer, M. "Techniques for Office Specification," Exxon Office Systems, New York, NY, April 1981.

18. Hammer, M. "The Functional Approach to Office Automation," Exxon Office Systems, New York, NY, April 1981.

19. Hammer, M. "Current Research: Office Taxonomy and Office Productivity," Exxon Office Systems, New York, NY, April 1981.

20. Hammer, M. "Topics in Office Information System Design," NCR Corporation, Columbia, SC, August 1980.

21. Hammer, M. "Toward Market and Customer Oriented Strategies in Office Automation," Citicorp, New York, NY, July 1980.

22. Hammer, M. "Understanding Office Automation," Nomura Study Tour on Office Automation, MIT Laboratory for Computer Science, Cambridge, MA, May 1981.

23. Ilson, R. "Etude: An Integrated and Interactive Text Processing System,"

    IBM San Jose Research Center, San Jose, CA, December 1980;

Summit Systems, Division of Vydec, December 1980;
Prime Computer Inc., Framingham, MA, January 1981.

24. Ilson, R. "Advanced Text Processing Systems," Graphic Design: Computers and Other Tools, Visible Language Workshop, MIT Architecture Department, Cambridge, MA, July 1980.

25. Kunin, J. "MIT Research in Office Automation," Office Automation Roundtable, New York, NY, April 1981.

26. Schoichet, S. "A Methodology for Office Analysis," MIT Sloan School of Management, Industrial Relations Section and Center for Informations Systems Research Seminar Series on Office Automation, Cambridge, MA, October 1980.

27. Schoichet, S. "An Approach to Office Systems Analysis," University of Southern California Information Sciences Institute, Santa Monica, CA, December 1980.

28. Schoichet, S. "Representing Formatted Documents,"

System Development Corporation, Santa Monica, CA, December 1980;
University of Southern California Information Sciences Institute Santa Monica, CA, December 1980.

29. Zdonik, S. "Databases in Office Automation," SIGMOD 1981, Ann Arbor, MI, May, 1981.

30. Zdonik, S. "Knowledge-Based Query Processing," VLDB-6, Montreal, Canada, October 1980.

Theses Completed

1. Barrett, F. "Communication Through Shared Data," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

2. Hirsch, F. "Knowledge Based Query Approximation," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 1981.

3. Ilson, R. "An Integrated Approach to Formatted Document Production,"

S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1980.

4. Schoichet, S. "Pagination in an Integrated and Interactive Document Composition System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

5. Stein, M. "Data-Independence Using a Role-Based Conceptual Schema," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1981.

Theses in Progress

1. Berkowitz, B. "An Interactive Page Makeup Facility for Formatting Office Documents," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

2. Bakopoulos, J. S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

3. Gilbert, E. S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

4. Good, M. "An Ease of Use Evaluation of an Editor and Formatter," S.M thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1981.

5. Kunin, J. "Analysis and Specification of Office Procedures," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

6. Niamir, B. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

7. Rosenstein, L. "Display Management in an Integrated Office Workstation," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge MA, Expected September 1981.

8. Sutherland, J. S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1982.

9. Zdonik, S. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

PROGRAMMING METHODOLOGY

Academic Staff

B. H. Liskov, Group Leader

Research Staff

P. R. Johnson

R. W. Scheifler

Graduate Students

T. Bloom
S.-Y. Chiu
C. Henderson
M. P. Herlihy
T. O. Humphries
J. Lancaster
J. E. B. Moss
B. Oki
J. C. Schaffert
E. W. Stark
W. E. Weihl

Undergraduate Students

E. T. Mueller
B. O'Connor
L.-S. Tong
K. A. Turkewitz

Support Staff

A. Rubin

Programming Methodology

1. INTRODUCTION

This year the Programming Methodology Group has continued work on the design of a programming language and system to support the construction and execution of distributed programs. We have refined our guardian model of distributed computation, and have concentrated on the problems of maintaining a consistent distributed state in the face of concurrent, potentially interfering activities, and in the face of system failures such as node crashes and network disruptions.

The approach we have taken is to extend the sequential language CLU with new linguistic mechanisms. Our design efforts are discussed in the following sections. Section 2 defines atomic activities and atomic objects, our fundamental mechanisms for achieving consistency, and presents the linguistic support for atomic activities. Section 3 describes the current structure of guardians. Finally, Sections 4 and 5 explore two of the most difficult problems in implementing the proposed system: dealing with distributed deadlocks, and dealing with processes whose locks have been broken, either through deadlock resolution, node crashes, or explicit aborts.

2. ATOMICITY

Our solution to the problem of maintaining a consistent distributed state is to make activities *atomic*. An activity can be thought of as a process that attempts to examine and transform a collection of objects from their current (initial) state to some new (final) state, with any number of intermediate state changes. Three properties distinguish an activity as being atomic: indivisibility, recoverability, and permanence. By indivisibility, we mean that the execution of one activity never appears to overlap (or contain) the execution of any other activity. If the objects being modified by one activity are observed over time by another activity, the latter activity will either always observe the initial states or always observe the final states, but it will never observe intermediate states. By recoverability, we mean that the overall effect of the activity is all-or-nothing: either all of the objects remain in their initial state, or all change to their final state. If a failure occurs while an activity is running, either it must be possible to complete the activity, or to restore all objects to their initial states. Finally, by permanence we mean that once an activity has completed, the changes it made to objects will not be lost subsequently (although later activities may make further changes to those objects). In practice, permanence is provided only to some reasonably high degree of probability, through the use of redundant or highly reliable storage devices [1].

In the remainder of this section we discuss our model of atomicity, and the supporting linguistic constructs. Our model is a slight variation of that developed by Eliot Moss, who completed a Ph.D. thesis on this topic during the year [2]

2.1. Actions

We call an atomic activity an *action*. An action may complete either by *committing* or *aborting*. When an action aborts, the effect is as if the action had never begun: all modified objects are restored to their previous state. When an action commits, all changes to objects are made permanent.

One simple way to implement the indivisibility property is to force actions to run sequentially. However, one of our goals is to provide a system that supports a fairly high degree of concurrency. The usual method of providing atomicity in the presence of concurrency, and the one we have adopted, is to guarantee *serializability* [3], namely, that the overall effect is as if the actions had been run sequentially in some order. To prevent one action from observing or interfering with the intermediate states of another action, we need to synchronize access to shared objects. In addition, to implement the recoverability property, we need to be able to undo the changes made to objects by aborted actions.

Since synchronization and recovery are likely to be somewhat expensive to implement, we do not provide these properties for all objects. In particular, objects that are purely local to a single action do not require these properties. The objects that do provide these properties are called *atomic* objects, and we restrict our notion of atomicity to cover only access to atomic objects. That is, atomicity is only guaranteed when the objects shared by actions are atomic objects.

Our implementation of atomic objects is based on a fairly simple locking model. There are two kinds of locks: read locks and write locks. Before an action uses an object, it must acquire a lock in the appropriate mode. The usual locking rules apply: multiple readers are allowed, but readers exclude writers and a writer excludes readers and all other writers. When a write lock is obtained, a (volatile) *version* of the object is made, and the action operates on this version. If the action ultimately commits, this version will be retained, and the old version discarded. If the action aborts, this version will be discarded, and the old version retained.

To ensure that the proper locks are obtained at the proper times, and that the versions are properly managed, atomic objects are encapsulated within *atomic* abstract data types. An abstract data type consists of a set of objects and a set of primitive operations; the primitive operations are the only means of accessing and manipulating the objects [4]. Atomic types have operations just like normal data types, except that each operation is implemented to obtain the appropriate locks and

make needed versions before manipulating the objects. The new language provides, as built-in types, atomic arrays, records, and variants, with operations nearly identical to the normal arrays, records, and variants provided in CLU. For example, atomic records have the usual component selection and update operations, but the selection operations obtain a read lock on the record (not the component), and the update operations obtain a write lock and create a version of the record the first time the action modifies the record. In addition, objects of built-in scalar types, such as characters and integers, are considered atomic, as are structured objects of built-in *immutable* types, such as strings, whose components cannot change over time.

All locks acquired by an action are held until the end of that action, a simplification of standard two-phase locking [5]. This rule avoids the problem of *cascading* aborts: if a lock on an object could be released early, and the action later aborted, any action that had observed the new state of that object would also have to be aborted.

Within the framework of actions, there is a straightforward way to deal with hardware failures at a node: they simply force the node to crash, which in turn forces actions to abort.[1] Since permanence is only required at the granularity of entire actions, acquired locks and changes to objects can be kept in volatile storage while an action executes, and hence may be lost due to node crashes. A node crash may take place after an action has visited the node, but before the action completes. To ensure that the action will abort, a two-phase commit protocol [6] is used. In the first phase, called the *prepare* phase, an attempt is made to verify that all locks are still held, and to record the new state of each modified object on a highly reliable storage device. Atomic stable storage [2] is used for this purpose.[2] If the prepare phase is successful, then in the second phase the locks are released, the recorded states become the current states, and the previous states are forgotten. If the prepare phase fails, the recorded states are forgotten and the action is forced to abort, restoring the objects to their previous states.

It has been argued that serializability is too strong a property for certain applications, and limits the amount of potential concurrency [7]. We believe that serializability is the desired property for most applications, *if* serializability is required only at the appropriate levels of abstraction. In particular, we have developed a mechanism for *user-defined* atomic data types. The important property of these types is that they are free to violate serializability internally, but they present an external interface that does not violate serializability. The objects defined by such a

---

[1] We assume that all failures can be detected as explained in [2].

[2] We need merely assume that stable storage is accessible to every node in the system; it is not necessary that every node have its own local stable storage devices.

type generally consist of some combination of atomic and non-atomic data. Process synchronization on non-atomic data is achieved through the use of *critical regions*. Each critical region has an associated object; no two processes may execute simultaneously in critical regions with the same associated object. In addition, while a process executes in a critical region the system is prevented from writing the object associated with that critical region to stable storage. Linguistic support for user-defined atomic types is still a subject of current research.

2.2. Nested Actions

Thus far, we have presented actions as monolithic entities. In fact, we provide hierarchically structured, *nested* actions. Nested actions, or subactions, are a mechanism for coping with failures, as well as for introducing concurrency within an activity. An action may contain any number of subactions, some of which may be performed sequentially, some concurrently. This structure cannot be observed from outside; i.e., the overall action still satisfies the atomicity properties. Subactions also appear as atomic activities with respect to other subactions. Subactions can commit and abort independently, and a subaction can abort without forcing its parent action to abort. However, the commit of a subaction is conditional: even if all subactions commit, aborting the parent action will abort all of the subactions. Further, permanence is only provided for top-level actions.

Nested actions aid in composing (and decomposing) activities in a modular fashion. For example, a collection of existing actions can easily be combined into a single, higher-level action, and can be run concurrently within that action with no need for additional synchronization. To extend this example, the concurrent actions might be reads or writes to the sites of a replicated data base. If only a majority of the reads or writes must be successful for the overall action to succeed, this is easily accomplished by committing the overall action as soon as a majority of the subactions commit, aborting all remaining subactions.

Nested actions have also been proposed by others [8] [9]; as previously mentioned, the model we have incorporated into the language is a simplification of the one developed by Moss [2]. To keep the locking rules simple, we do not allow a parent action to run concurrently with its children. The rule for read locks is extended so that an action may obtain a read lock on an object provided every action holding a write lock on that object is an ancestor. An action may obtain a write lock on an object provided every action holding a (read or write) lock on that object is an ancestor. When a subaction commits, its locks are inherited by its parent; when a subaction aborts, its locks are discarded.

Note that the locking rules permit multiple writers, which implies that multiple

versions of objects are now needed. However, since writers must form a linear chain when ordered by ancestry, and actions cannot execute concurrently with their subactions, only one writer can ever actually be executing at one time. Hence, it suffices to use a stack of versions (rather than a tree) for each atomic object. On commit, the top version becomes the new version for the parent; on abort the top version is simply discarded. A detailed description of locking and version management in a system supporting nested actions is presented in [2].

Permanence is only provided for top-level actions, so only one version of an atomic object must be kept on stable storage; the stack of versions can be kept in volatile storage. Further, the two-phase commit protocol is only needed for top-level actions. In fact, when a subaction commits or aborts, it is not even necessary to distribute this information to the nodes visited by the action. The information could only affect the ability of concurrent siblings of the subaction to acquire locks on the same objects locked by the subaction. This can be handled by *querying*. Details of a query mechanism are presented in [2].

2.3. Remote Procedure Call

Nested actions help in coping with communication failures, and this is perhaps their single most important application. Logical nodes (described in the next section) in our system communicate via messages. We believe that the most desirable form of communication is the paired send and reply: for every message sent, a reply message is expected. In fact, we believe the form of communication that is needed is *remote procedure call*, namely, that (effectively) either the message is delivered and acted on exactly once, with exactly one reply received, or the message is never delivered and the sender is so informed.

The rationale for the high-level, at-most-once semantics of remote procedure call is presented in [10]. Briefly, we believe the system should mask from the user low-level issues, such as packetization and retransmission, and that the system should make a reasonable attempt to deliver messages. However, we believe the possibility of long delays and of ultimate failure in sending a message cannot and should not be masked. The sender should be allowed to cope with communication failure according to the demands of the particular application, and must be able to terminate communication if the delays become excessive. If communication is terminated, then the remote procedure call should have no effect.

The all-or-nothing nature of remote procedure call is quite similar to the recoverability property of actions, and the ability to cope with communication delays and failures is quite similar to the ability of an action to cope with the failures of subactions. Therefore, it seems natural to implement a remote procedure call as a

subaction: communication failures will force the subaction to abort, and the sender has the ability to abort the subaction on demand. However, as mentioned above, aborting the subaction does not force the parent action to abort. The sender is free to find some other means of accomplishing its task, such as communicating with some other node.

2.4. Remarks

In our model, there are two kinds of actions: nested actions and top-level actions. We believe these correspond in a natural way to activities in the application system. Top-level actions correspond to activities that interact with the external environment. For example, in an airline reservation system, a top-level action might correspond to an interaction with a clerk who is entering a related sequence of reservations. Nested actions, on the other hand, correspond to internal activities that are intended to be carried out as part of an external interaction; a reservation on a single flight is an example.[3] Atomic types provide two services to the user of the language: they guarantee serializability and they automatically undo effects of aborted actions. The user of our language does not need to write any code to undo or compensate for the effects of aborted actions. On the other hand, the commit of a top-level action is irrevokable. If that action is later found to be in error, actions that compensate for the effects of the erroneous action, and all later actions that depended on it (read its results), must be defined and executed by the user. Note that in general there is no way that such compensation could be done automatically by the system, since extra-system activity is needed (e.g., canceling already issued checks).

Given our use of a locking scheme to implement atomic objects, it is certainly possible for two (or more) actions to *deadlock*, each attempting to acquire a lock held by the other. Although in many cases deadlock can be avoided with careful programming, certain deadlock situations are unavoidable. Our method of breaking deadlocks is to abort actions, rather than refuse locks. The system is not guaranteed to detect deadlocks; in general, deadlocks must be broken by timing out and aborting actions. However, Moss [2] has developed a distributed deadlock detection algorithm that could be used to detect a large class of deadlocks. This algorithm is discussed in Section 4.

---

[3] Nested top-level actions are also available. They are useful for accomplishing benevolent side effects, e.g., updating a cache or performing garbage collection, that need not be undone if the parent aborts.

2.5. Linguistic Support for Actions

Top-level actions are created by means of the action statement:

enter topaction *body* **end**

This causes the *body* to execute as a new top-level action. When the *body* completes, it does so either by committing or aborting. It is also possible to have an inline subaction:

enter action *body* **end**

This causes the *body* to run as a subaction of the action that executes the **enter**.

Inline actions can terminate in many different ways. In all cases, they must indicate when terminating whether they are committing or aborting. Since committing is assumed to be most common, it is the default; the qualifier **abort** can be prefixed to any termination statement to override this default. For example, an inline action can execute

leave

to commit and cause execution to continue with the statement following the **enter** statement; to abort and have the same effect on control, it executes

abort leave

Falling off the end of the *body* causes the action to commit.

A group of concurrent subactions are created by means of the action statement:

coenter {*coarm*} **end**

where

coarm :: = *armtag*[**foreach***decl-list***in***iter-invocation*]
body

armtag :: = **action** | **topaction**

A **foreach** clause indicates that multiple instances of the coarm will be activated, one for each item (a collection of objects) yielded by the given iterator invocation.[4]

---

[4] An iterator is a limited kind of coroutine that provides results to its caller one at a time [11] [12].

Each such coarm will each have local instances of the variables declared in the *decl-list*, and the objects constituting the yielded item will be assigned to them. Execution of the **coenter** starts by running each of the iterators to completion, sequentially, in textual order. Then all coarms are started simultaneously as concurrent siblings. Each coarm instance runs in a separate process, and each process executes within a new top-level action or subaction, as specified.

A simple example making use of **foreach** is

```
coenter action foreach i: int in int$from_to (1, 5)
    p(i)
    end
```

which creates five processes, each with a local variable *i*, having the value 1 in the first process, 2 in the second process, and so on. Each process runs in a newly created subaction.

A coarm may terminate without terminating the entire **coenter** either by falling off the end of its *body*, or by executing a **leave** statement. As before, **leave** may be prefixed by **abort** to cause the completing action to abort; otherwise the action commits.

A coarm also may terminate by transferring control outside the **coenter** statement. Before such a transfer can occur, all other active coarms of the **coenter** must be terminated. To accomplish this, the system forces all coarms that are not yet completed to abort. To abort a coarm, the system waits for its process to leave any critical regions (see Section 2.1); it then destroys the process and aborts the action.

A simple example where such early termination is useful is in timing out communication with another node:

```
coenter
  action    remote procedure call
            exit done
  action    wait for some amount of time
            exit timed_out
  end
```

Whichever of these two actions completes first, it commits itself and aborts the other. In either case, the abort takes place immediately (since there are no critical regions). In particular, it is not necessary for the reply to the remote procedure call to be received before the sending action can be aborted. This last fact is important, since the reason for timing out may be to avoid waiting a long time due to crashes, loops,

or deadlocks elsewhere in the system. However, such timeouts can lead to *orphan* processes, as described in Section 5.

There is another form of **coenter** for use outside of actions, as in the **recover** and **start** sections of a guardian (see the next section). In this form the *armtag* is **process**. The semantics is as above, except that no actions are created.

3. GUARDIANS

In our language, a distributed program is composed of a group of *guardians*. A guardian encapsulates one or more resources, and provides controlled access to those resources. The external interface of a guardian consists of a set of operations called *handlers*, which may be invoked by other guardians using the remote procedure call semantics discussed previously. The guardian executes the calls on these handlers, synchronizing them as needed. Furthermore, it may refuse to perform an access desired by a caller if the caller does not have proper authorization.

Internally, a guardian contains data objects and processes. Some of the data objects comprise the global state of the guardian; these objects, such as the actual resources, are shared by the processes. Other objects are local to the individual processes.

Guardians exist entirely at a single physical node: all of a guardian's processes run at that node, and (the volatile state of) the guardian's objects are stored at that node. However, a guardian survives crashes of the node at which it resides.

Some of the objects in the guardian state are *stable*; these are the objects written to stable storage when top-level actions commit. After a crash of the guardian's node, the language support system re-creates the guardian with the stable objects as they were when last written to stable storage. A process is started in the guardian to re-create that portion of the guardian state that is not stable. This *volatile* state can be used to hold redundant information, e.g., an index for fast access into a data base. Once the volatile state has been restored, the guardian can resume background tasks, and can respond to new requests.

Although the processes inside a guardian can share objects directly, direct sharing of local objects between processes in different guardians is not permitted. The only method of inter-guardian communication is by invoking handlers, and the arguments to handlers are passed by value: it is impossible to pass a reference to a local object in a message. This rule ensures that objects local to a guardian remain local, and thus ensures that a guardian retains control of its own objects. It also

provides the programmer with a concept of what is expensive: local objects are close by and inexpensive to use, while non-local objects are more expensive to use; this is underlined by the different access methods (procedure call versus handler call). A method for passing data values between heterogeneous nodes using different internal representations is presented in [13]

Guardians and handlers are an abstraction of the underlying hardware of a distributed system. A guardian is a logical node of the system (several guardians may reside at the same physical node), and inter-guardian communication via handlers is an abstraction of the physical network. The most important difference between the logical system and the physical system is reliability: the stable state of a guardian is never lost (to a very high probability), and the remote procedure call semantics of handler calls ensures that handlers either succeed completely or have no effect.

3.1. Guardian Structure

The current syntax of a guardian definition is shown in Figure 1.[5] A guardian definition implements a special kind of abstract data type whose operations are handlers. The name of this type, and the names of the handlers, are listed in the guardian header. In addition, the type provides a *create* operation that is invoked to create new guardians of the type. Guardians may be *parameterized*, providing the ability to define a class of related abstractions by means of a single module. Parameterized types are discussed in [11] [12].

---

[5] In the syntax, optional clauses are enclosed with [ ], zero or more repetitions are indicated with { }, and alternatives are separated by |.

```
name = guardian [[ parameter-decls ]]is handler-names

    {[stable] variable-decls-and-inits}

    [init[argument-decls][signals (exceptions)]
        body
        end]

    [recover body end]

    [start body end]

    [handler-definitions]

    % local procedures and iterators may also be defined

    end name
```

Figure 1. Guardian Structure.

The first internal part of a guardian is a list of variable declarations, with optional initializations, defining the guardian state. Some of these variables can be declared as **stable** variables; the others are volatile variables.

The stable state of a guardian consists of all objects *reachable* from the stable variables; these objects, called stable objects, have their new versions written to stable storage by the system when top-level actions commit. The new language, like CLU, has an object oriented semantics. Variables name (or refer to) objects residing in a free storage area. Objects themselves may refer to other objects, permitting recursive and cyclic data structures without the use of explicit pointers. The set of objects reachable from a variable consists of the object that variable refers to, any objects referred to by that object, and so on. (In a language with explicit pointers, the concept of reachability would still be needed to accommodate the use of pointers in stable objects.)

We require that all stable objects also be atomic objects, as discussed in Section 2. This requirement is enforced by compile-time type-checking: the type of each stable variable must be atomic. One reason for this requirement is that the system knows how to synchronize with activity in the guardian to ensure that atomic objects are written to stable storage in internally consistent states. In addition, the system knows how to write atomic objects in an incremental manner and still preserve the sharing among these objects. These same properties do not hold for non-atomic objects. As mentioned in Section 2, the language provides a number of built-in

atomic types, and users may define new abstract atomic types. In fact, guardians are themselves one class of user-definable atomic types.

The next part of the guardian definition is the **init** section. Guardian instances are created dynamically. The **init** section, and any initializations attached to the variable declarations of the guardian state, are run whenever an instance of the guardian is to be created. This code, which executes as a subaction of the caller, initializes both the stable and volatile guardian state. If **init** terminates successfully (by returning or falling off the end), the guardian's creator (a process in some other guardian) receives the guardian object and can then invoke its handlers. If **init** terminates by signalling an exception (see [14]), guardian creation fails and the exception is propagated back to the creator.

To create a new guardian, the *create* operation is called. This operation takes the same formal arguments as the **init** section, and either returns a guardian object or signals one of the exceptions listed in the **init** section. For example, suppose we have a guardian definition with header:

```
g = guardian is h1, h2, h3
```

and some process executes

```
x: g := g$create (...)
```

The guardian object *x* is created at the same physical node where the process is executing. The handlers provided by the guardian may be referred to as *x.h1, x.h2* and *x.h3*.

The **recover** section runs after a crash. Before creating a process to run the **recover** section, the system restores the guardian's stable state from stable storage. Since updates to stable storage are made only when top-level actions commit, the stable state has the value it had at the latest commit of a top-level action before the guardian crashed. Actions that had executed at the guardian prior to the crash, but had not yet fully committed, are aborted, and their changes to the stable state are lost.

The job of the **recover** section is to re-create the volatile state that is consistent with the stable state. This may be trivial, e.g., creating an empty cache, or it might be a lengthy process, e.g., creating a data base index. The **recover** section is not run as an action, although it may create top-level actions, as explained in Section 3.4.

After the successful completion of **init** (when the guardian is first created) or **recover** (after a crash), two things happen inside the guardian: a process is created to run the **start** section, and handler invocations may be executed. The **start**

section provides a means of performing periodic (or continuous) tasks within the guardian. Like the **recover** section, the **start** section is not run as an action.

3.2. Handlers

Handlers, like procedures in CLU, are based on the termination model of exception handling [14]. A handler can terminate in one of a number of conditions: one of these is considered to be the "normal" condition, while others are "exceptional," and are given user-defined names. Results can be returned both in the normal and exceptional cases; the number and types of results can differ among conditions. The header of a handler definition lists the names of all exceptional conditions and defines the number and types of results in all cases. For example,

```
file_date = handler (fn: file_name) returns
        (date) signals (not_possible(string))
```

is the header of a handler whose calls either terminate normally, returning a result of type date, or exceptionally in condition *not_possible* with a string result. In addition to the named conditions, any handler can terminate in the *failure* condition, returning a string result; failure termination may be caused explicitly by the user code, or implicitly by the system when something unusual happens, as explained further below.

Handler calls differ from ordinary procedure calls in several important ways:

1) Procedures always run inside the guardian in which they are called. Handlers usually belong to some other guardian (although a call to a handler of your own guardian is permitted), and that guardian is likely to reside on some other node. Thus, the system will construct a message containing the arguments and send it to the appropriate node. When the handler call terminates, the system constructs another message containing the termination condition and results, and sends it back to the calling guardian.[6]

2) Procedure arguments and results are passed *by sharing* (see [12]); i.e., the argument and result objects are shared between the calling and called procedure. As mentioned above, handler arguments and results are always passed by value.

---

[6] If the calling and called guardians reside on the same node, the system may be able to optimize this message passing.

3) The body of a procedure is executed in the same process that invoked the procedure. When a handler is invoked, the calling process stops running, and a new process is created at the guardian of the handler. This process runs the handler to completion (until a **return** or **signal**), and is then destroyed. The calling process continues running when the result message is received.

4) Handlers are executed as subactions of the calling action. Procedures simply execute within the calling action. Handlers are executed as subactions to achieve the remote procedure call semantics previously discussed.

Since a handler executes as an action, it must, in addition to returning or signalling, either commit or abort. We expect committing to be the most common case, and therefore execution of a **return** or **signal** statement indicates commitment. To cause an abort, the **return** or **signal** is prefixed with **abort**.

Let us examine a step-by-step description of what the system does when a handler is invoked:

1) A new subaction is created.

2) A message containing the arguments is constructed. Since part of building this message involves executing user-defined code (see [13]), message construction may fail. If so, the subaction aborts and the call terminates with a *failure* exception.

3) The system sends the message to the target guardian. If the handler's guardian no longer exists the subaction aborts and the call terminates with a *failure* exception.

4) The system makes a reasonable attempt to deliver the message, but success is not guaranteed. The reason is that it may not be sensible to guarantee success under certain conditions, such as a crash of the target node. In such cases, the subaction aborts and the call terminates with a *failure* exception. The meaning of such a failure is that there is very low probability of the call succeeding if it is repeated immediately. Hence, there is no reason for user code to repeatedly retry handler calls. Rather, user programs should guarantee progress by retrying top-level actions, which may fail because of node crashes even if all handler calls succeed.

5) The system creates a process at the receiving guardian to execute the

handler. Note that multiple instances of the same handler may execute simultaneously. The system takes care of locks and versions of atomic objects used by the handler in the proper manner, according to whether the handler commits or aborts. When the handler terminates, the system destroys the process.

6) The system creates the response message and sends it to the calling guardian. If this is impossible (as in (2) or (4) above), the subaction aborts and the call terminates with a *failure* exception.

7) The calling process continues execution. Its control flow is affected by the termination condition as explained in [14]. For example, for a call of file - date above we might have

```
d: date := file_date(fn)    % normal return
    except when not_possible,
           failure (why: string): ... % exceptional return
       end
```

3.3. Remarks

Guardians maintain complete local control over their local data. The data inside a guardian is truly local; no other guardian has the ability to access or manipulate the data directly. The guardian provides access to the data via handler calls, but the actual access is performed inside the guardian. It is the guardian's job to guard its data in three ways: by synchronizing concurrent access to the data, by requiring that the caller of a handler have the authorization needed to do the access, and by making enough of the data stable so that the guardian as a whole can survive crashes without loss of information.

While guardians are the unit of modularity, actions are the means by which distributed computation takes place. A top-level action will start at some guardian. This action can perform a distributed computation by making handler calls to other guardians; those handler calls can make calls to still more guardians, and so on. Since the entire computation is an atomic action, it is guaranteed that the computation is based on a consistent distributed state, and that when the computation finishes, the state is still consistent, assuming in both cases that user programs are correct.

To provide this guarantee, the system must do a lot of work. It keeps track of the history of actions: which guardians are visited, which objects are read, and which are modified. As subactions commit and abort, this history is modified appropriately.

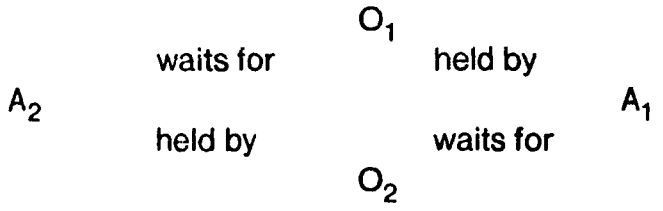
Finally, when a top-level action commits, this history is used to ensure that none of the guardians involved[7] have crashed since they were used. If this condition is met, the system updates stable storage appropriately, releases locks, and discards old versions. If the condition is not met, the system forces the action to abort, releases all locks, and restores old versions.

4. DISTRIBUTED DEADLOCK DETECTION

In this section we present an algorithm developed by Moss [2] for detecting deadlocks in a distributed system where read/write locks are used to implement atomic objects. The algorithm will detect any deadlock involving a set of actions, each of which is waiting on a single lock of a built-in atomic object. It will not detect more complicated deadlocks, such as those where the actions are (busy) waiting on user-defined logical locks or on complex conditions.

There are essentially three ways of handling deadlocks: prevention, avoidance, and detection. Prevention, i.e., guaranteeing that deadlocks can never arise in the first place, is virtually impossible in the context of a general programming language like ours. Avoidance schemes typically work by assigning priorities to actions; when a higher priority action waits on a lock held by a lower priority action, the lower priority action is aborted after some time period. Detection schemes allow deadlocks to occur and try to find and resolve them after the fact. The advantage of avoidance schemes over detection schemes is their relative simplicity; the disadvantage of avoidance schemes is that actions may be aborted when in fact there is no deadlock.

An example of a simple form of deadlock involves two actions, A_1 and A_2, and two objects, O_1 and O_2, where A_1 holds a read lock on O_1, A_2 holds a read lock on O_2, and A_1 attempts to get a write lock on O_2 while A_2 attempts to get a write lock on O_1:



The general form of deadlocks detected by this algorithm involve similar cycles of alternating actions and objects. In essence, deadlock detection consists of finding

---

[7] The guardians involved are those visited by handler calls performed as subactions of the top-level action, where the subaction and all of its ancestors have committed.

cycles in a directed graph, where the nodes are actions and objects, and the edges indicate objects locked by actions and actions waiting for locks on objects. In a distributed system such as ours, the problem is complicated by the fact that no single node knows about the entire graph; it is necessary to communicate information between nodes in order to guarantee all cycles will be found.

The algorithm to be described is of the edge-chasing variety. The graph is never actually built completely. Rather, individual paths through the graph are traced; if a path ever closes on itself, a deadlock has been found. The algorithm makes use of priorities: every action must have a unique priority, and the priority of a given action must be higher than the priority of each of its subactions. A simple scheme satisfying this property is to use the start time of the action (made suitably unique) as its priority, with the obvious ordering.

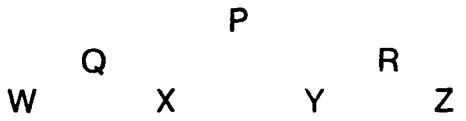
Since a deadlock cycle can form in many different ways, it may seem that deadlock detection must be initiated whenever an action waits for a lock. However, we can avoid this by using priorities in the following way. For a given deadlock cycle, we can eliminate the objects involved and compress the cycle into one showing only action dependencies. For example:

$$A_1 \quad A_2 \quad A_3 \quad \ldots \quad A_n$$

Given that all priorities are unique, one of these actions, say A_i, must have a lower priority than the others. We would like an algorithm that only initiates deadlock detection once for a given cycle, and a natural time for that to occur is when A_{i-1} waits for a lock held by A_i. Of course, there is no way of knowing, for any particular dependency, whether the action with minimum priority is involved. Instead, we approximate this by initiating detection whenever a higher priority action waits for a lock held by a lower priority action, and we will introduce a mechanism to terminate the detection if we discover that the lower priority action is not the minimum priority action.

So far, we have ignored a complication arising from nested actions. When an action X waits for a lock held by Y, it is not sufficient to compare the priority of X against the priority of Y. In the case where X and Y are completely unrelated actions, X cannot obtain the lock until the top-level action containing Y has committed, due to our lock inheritance rules, so we should in fact compare the priority of X against the priority of this top-level action. More generally, X cannot obtain the lock until the oldest ancestor of Y that is not also an ancestor of X has committed.

For example, given the following action structure,



if X is waiting for a lock held by Y, X cannot obtain the lock until R commits. Hence we should compare the priority of X against the priority of R. However, note that in some sense Q is also waiting for the lock held by Y, since Q cannot commit until X commits and X is waiting for that lock. Further, the priority of Q may be greater than the priority of R even if the priority of X is less than the priority of R. To ensure that detection is initiated for all deadlocks, we should use the priority of Q rather than the priority of X. That is, we should compare the priority of the oldest ancestor of X that is not also an ancestor of Y against the priority of the oldest ancestor of Y that is not also an ancestor of X. For brevity in the remainder of this section, we define OANA(A, B) to be the oldest ancestor of A that is not also an ancestor of B.

To summarize so far, when action X waits for a lock held by Y, we initiate detection if the priority of OANA(X, Y) is greater than the priority of OANA(Y, X). Detection proceeds by sending a *detect* message to the node where R = OANA(Y, X) was created. A detect message consists of a list of actions waiting for locks, the assumed minimum priority action holding one of those locks, and the current action in question. The list of actions corresponds to a path through the dependency graph, with each action in the list waiting for a lock held by the next action (or a descendant of the next action) in the list. The first detect message thus contains the singleton list <X>, the assumed minimum priority action R, and the current action in question, also R.

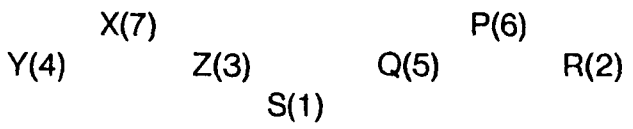
When a detect message concerning an action R is received at a node, the system first checks to see if that action is waiting for a lock. If it is, and that lock is held by an action S, then the system checks to see if T = OANA(S, R) is one of the actions (or an ancestor of one of the actions) in the list of actions sent in the detect message. If so, a deadlock has been found, and some action, such as the minimum priority action, can be aborted to break the deadlock. If a deadlock is not found, the system checks to see if the priority of T is less than the priority of the assumed minimum priority action sent in the detect message. If it is, then the assumed minimum priority action is not actually minimal, and detection can be terminated. If the priority is greater, then a detect message for T is sent to the node where T was created, with the action R appended to the list of actions waiting for locks.

If, on the other hand, a detect message for action R is received and the action is not waiting for a lock, then a detect message is sent to each running subaction of R, with the same list of waiting actions as in the original detect message. In this way,

detection proceeds down the action hierarchy until actions waiting for locks are found.

Note that replies to detect messages are not needed. In fact, the remote procedure call semantics discussed earlier is not used at all for detect messages. Rather, the initiator of a detection scan simply retransmits the initial detect message periodically until the desired lock becomes available.

As a simple example, suppose we have the following action structure, with priorities given in parenthesis:



Suppose that S is waiting for a lock held by R, R is waiting for a lock held by Y, and Z is waiting for a lock held by Q. Note that, as stated, there is no direct cycle in the dependency graph, although there is a deadlock. Also note that each action waiting for a lock has a priority less than the priority of the action holding that lock. Thus, the comparing the priorities of oldest ancestors is critical to detecting this deadlock.

We will examine the detect messages which result when action S waits for the lock held by R, since R is the minimum priority action. Since the priority of Q = OANA(S, R) is greater than the priority of R = OANA(R, S), detection is initiated by sending a detect message with action list <S> and minimum priority action R to the node where R was created. Since R is waiting for a lock held by Y, and the priority of X = OANA(Y, R) is greater than the priority of the minimum priority action R, a detect message containing the action list <S, R> is sent to the node where X was created. X is not waiting for a lock, so detect messages are sent for the children of X, namely Y and Z. The action list in both of these detect messages is again <S, R>. The detect message sent for Y will be ignored, since Y is not waiting for a lock and does not have any children. However, when the detect message for Z is received, the system will notice that Z is waiting for a lock held by Q, and that Q is an ancestor of an action (S) in the list of actions sent in the detect message. Thus, a deadlock has been found.

5. ORPHANS AND INTERNAL CONSISTENCY

In this section we deal with some implementation issues surrounding the general problem of broken locks. The locks held by an action can be broken by an *active* abort of an ancestor action, initiated either by the application program or by the system as part of deadlock resolution, and locks can also be broken by a *passive*

abort of an ancestor action, as the result of a node crash. Broken locks can result in *externally* inconsistent behavior, where the system claims it has done something when it has not, or in *internally* inconsistent behavior, where individual actions may observe inconsistent states of data.

It is not difficult to ensure external consistency; it suffices to guarantee that an action is not allowed to commit to the top-level if any of its locks have been broken. This guarantee is provided through the use of the two-phase commit protocol mentioned in Section 2, where the system verifies that all of the action's locks are still intact before the final commit takes place. Note that external consistency does not include communication outside the system (e.g., output to physical devices) that cannot be undone when actions are aborted; it is up to the programmer to avoid this sort of inconsistency.

Ensuring internal consistency is much harder. We can divide the problem into two parts: dealing with actions when their ancestors have been actively aborted (either by the application or by the system), and dealing with passive aborts resulting from node crashes. These subproblems are dealt with separately below.

5.1. Orphans

An *orphan* is a process, executing on behalf of some action, that continues to run after an ancestor action has been actively aborted. A simple example of where orphans arise is in communication between guardians. As mentioned in Section 2.5, an action waiting for a reply to a handler call may be timed out and aborted without actually waiting for the remote process executing the handler to be terminated.

Orphans cause two kinds of problems. One is a simple waste of resources: the work done by an orphan is not wanted, yet the orphan is using resources to do it. The other problem is that, from the viewpoint of the semantics of the language, the completion of an action implies that all subactions of that action have also been completed. Thus, the system must ensure that all activity on behalf of those subactions *appears* to have ceased, even if in reality it has not.

We have developed two algorithms for dealing with orphans. The first algorithm guarantees that all activity on behalf of an action and its subactions does in fact cease when the action aborts. The second algorithm merely guarantees that the system behaves in a manner that is semantically equivalent to waiting for all such activity to cease.

The first algorithm is based on *reassurance* messages. While a handler is being executed at a guardian, the manager of that guardian requires periodic reassurance from the manager of the calling guardian that the execution should not be aborted.

Reassurance is given by sending a reassurance message every R time units. These messages require no reply, and are not sent using the remote procedure call semantics. A reassurance message must be received at least every W time units or the the process executing the handler will be terminated and the handler subaction will be aborted. To minimize the probability of unnecessary aborts, R must be considerably less than W. We assume R and W can be chosen so that the time required to send a message is much less than R.

When a guardian manager has failed to receive reassurance for a particular handler call, it must not only abort the handler subaction, but abort any nested handler calls made by the handler to other guardians. This can be accomplished simply by ceasing to send reassurance messages for those handler calls. There is a problem, however, because for any particular handler call the depth of nested handler calls is completely unknown. Hence, in order to abort a handler call, the system must wait on the order of NW time units, where N is the number of guardians in the system, in order to guarantee that all activity has ceased.

We can reduce the waiting period to within reasonable bounds by establishing a maximum depth on the nesting of handler calls. The maximum depth is sent along as part of each handler call, and the maximum depth sent with each nested call is one less than the maximum depth for the current call. In this way, the waiting time to abort a handler call is on the order of DW time units, where D is the maximum depth sent with that call. From our experience with nesting of procedure calls in CLU, we expect the nesting of handler calls will usually be quite shallow, so a small value for D can probably be chosen that will work in most cases without making the waiting time too large.

Of course, occasionally this depth will be insufficient. When this occurs, the system will increase the depth automatically. When the depth is exceeded for a particular handler call, permission to increase the depth will have to be obtained from the manager of the guardian where the handler was invoked. If that invocation is nested inside another handler call, then the manager will in turn have to obtain permission from the manager of the guardian where that call originated, and so on up the call chain.

One way to terminate a handler process is to crash the guardian. However, if there are other actions running concurrently at the guardian, or committed subactions whose parent actions have yet to commit, crashing the guardian will cause more actions to abort. Fortunately, it is generally possible to simply terminate the process executing the handler, as well as any subprocesses executing local subactions of that handler. A process can be terminated immediately unless it is in a critical region (see Section 2.5). Even if the process is in a critical region, the system can wait for the process to exit the region and then force termination.

For the algorithm to work, however, we must place a bound on the time it takes to terminate a process and all of its local subprocesses, once all of its non-local (nested) handler calls have been aborted. If this bound is K time units, then the waiting time to abort a handler call is DW + K. Since the amount of time a process can remain in a critical region is unbounded, there being no restrictions at the language level, the system can wait K time units for the processes to exit all critical regions, but after K time units the only option is to crash the guardian. Potentially this is a very serious defect in the algorithm, although it is difficult to evaluate how often such situations will arise in practice.

The second algorithm is based on a kind of lazy evaluation, recognizing that it is not necessary to eliminate an orphan until there is a potential conflict between that orphan and an action that might know the orphan should have been terminated. The basic idea is that, at every guardian, before execution of a handler is allowed to begin, any currently running actions that the handler might recognize as orphans must be stopped. In addition, once execution of the handler commences (and even after it has completed), no new actions that could be (or could have been) recognized as an orphan by that handler can be allowed to start execution.

To accomplish this, the system must keep track of all aborts, and must propagate abort information to other parts of the system at least as fast as actions can propagate equivalent information. The most obvious way to do this is to piggy-back abort information on every send and reply of handler calls.

Each guardian manager maintains, for each action that executes (or has executed) at that guardian, a volatile list of subactions (and descendants of subactions) that have been actively aborted. Whenever an action waiting for a reply to a handler call is aborted at a guardian, the action is added to the abort list for its parent action. All local activity on behalf of the action (or any of its subactions) must be stopped before the parent action can continue. Whenever an action commits, the abort list for that action is merged into the list for its parent action. Each manager also maintains a stable list of top-level actions for which orphans might exist. A top-level action is added to this stable abort list, during the prepare phase of its two-phase commit, if any of its descendants were actively aborted.

Whenever a guardian manager sends a handler call request, it also transmits the abort list of the sending action, as well as the abort lists of all ancestors of the sending action. Whenever a manager receives a handler call request, it first checks the sending action against its stable top-level abort list, and against the volatile abort lists of any known ancestors of the sending action. If the sending action is in one of these lists, or an ancestor of the sending action is in one of these lists, the sending action is an orphan. In this case, the manager transmits a special reply to that effect, and the sending manager must abort the orphan. If the sending action is not an

orphan, then all currently executing actions are checked against the abort lists sent in the request, and any that are orphans must be aborted before the handler can be executed.[8] The guardian manager then merges the abort lists sent in the request with its existing abort lists.

Whenever a guardian manager replies to a handler call, it also transmits the abort list for that action. Whenever a manager receives a reply to a handler call, all currently executing actions are checked against the abort list sent in that reply, and any that are orphans must be aborted before the reply can be passed on to the sending action. The manager then merges the abort list with the abort list for the parent action.

Finally, as part of the prepare phase of a two-phase commit, the manager of each guardian involved must abort all currently executing subactions of the given top-level action, since they are of necessity orphans. As stated above, the top-level action is added to the stable abort list at each manager if any of its descendants were actively aborted.

The basic problem with this algorithm is that actions in the stable abort lists are never removed, so these lists (of which there must also be volatile copies for searching purposes) will grow quite large. In theory, removing actions from these lists is difficult. The only way to remove an action is to guarantee that it has no orphans running anywhere in the system, which requires communicating with all nodes. In practice, however, an action can probably be removed safely after some fixed period of time, such as several days or weeks. Actions could be stamped with the start time of their top-level ancestor, and handler call requests could be refused if they derive from a top-level action that is too old. Once the time limit for a top-level action has expired, the action can be removed from the stable abort list after a short time (viz., the maximum skew between clocks in the system).

5.2. Crashes

Dealing with an action, some of whose relatives have been aborted due to node crashes, also requires substantial work on the part of the system. The following is a simple example of the problem. Suppose we have a guardian G containing an atomic record with a single integer component. Suppose that value is currently 2. Now suppose we have two actions, R and W, that run at some other guardian but perform handler calls to G. Action R first reads the value in the atomic record with

---

[8]Note that this is still only a local abort; if the orphan is waiting for a reply to a handler call, it is not necessary to wait for the handler to terminate.

one handler call, and then reads the value again in a second handler call, and checks that the two values are the same. Action W writes the value 3 into the record. If the node containing G crashes at an inoppertune time, we get the following:

R reads the value 2 at G
G crashes and comes back up, forgetting R's lock
W writes the value 3 at G
W commits
R reads the value 3 at G

To preserve internal consistency, R must be aborted either when its call is received at G, or (at the latest) when R attempts to acquire the read lock on the atomic record at G. One way to achieve this is to send along with every handler call request a list of all guardians visited so far by the sending action (and its ancestors). This list must be checked against the volatile list kept by the guardian manager of every action that has visited the guardian. If the incoming list indicates that the sending action (or an ancestor) has previously visited the guardian, but the list kept by the manager does nct contain the same information, then the sending action must be aborted.

Unfortunately, this relatively simple algorithm fails to ensure internal consistency in some cases. For example, suppose we now have two guardians, G1 and G2, each containing an atomic record with a single integer component. The constraint on these guardians is that the two records must contain the same integer value. Suppose that value is currently 2. Now suppose that we again have two actions, R and W. Action R first reads the value at G1, then reads the value at G2, and checks that they are the same. Action W first writes the value 3 at G1 and then writes the value 3 at G2. If the node containing G1 crashes at an inopportune time, we get the following:

R reads the value 2 at G1
G1 crashes and comes back up, forgetting R's lock
W writes the value 3 at G1
W writes the value 3 at G2
W commits
R reads the value 3 at G2

Note that R will eventually abort because its two-phase commit will discover that the lock at G1 has been broken. Hence, external consistency is at least preserved. However, internal consistency has been violated, and the algorithm sketched above will not detect this inconsistency.

Not all systems undertake to preserve complete internal consistency. Instead, the only guarantee is that, for the duration of a given action, the state of each individual

object used by that action will not appear to be changed by any other action; no guarantee is given about higher-level, implicit consistency constraints between objects. This weaker guarantee is satisfied by the above algorithm. However, we believe internal consistency is an important property that the system must guarantee.

To deal with this problem, we introduce *crash counts.* Each guardian manager keeps a stably recorded counter that is incremented after every crash. In addition, each manager maintains a map pairing guardians with their last known crash counts. Whenever a guardian manager sends a handler call request, it also transmits a list of all guardians visited so far by the sending action (and its ancestors), together with their last known crash counts. Whenever a manager receives a handler call request, it first checks the incoming list of guardians and crash counts against its current crash map. If the incoming list contains a guardian with a crash count that is less than the corresponding crash count in the current map, then the sender must be aborted, and a special reply to that effect is transmitted. If the incoming list contains a guardian with a crash count that is greater than the corresponding crash count in the current map, then all currently executing actions that have visited that guardian (or whose ancestors have visited that guardian) must be aborted before the handler can be executed.[9] The manager then merges this new crash count information into its map.

Whenever a manager replies to a handler call, it also transmits a list of all guardians visited by that action (and its descendants), together with their last known crash counts. Whenever a manager receives a reply to a handler call, this incoming list is compared against the current map. If the incoming list contains a guardian with a crash count that is greater than the corresponding crash count in the current map, then all currently executing actions that have visited that guardian (or whose ancestors have visited that guardian) must be aborted before the reply is passed on to the sending action. The manager then merges this new crash count information into its map.

Finally, as part of the prepare phase of every two-phase commit, the manager of each guardian involved must stably record its current crash count map, to be used as the initial map after a crash. In addition, the prepare phase message must include a list of all guardians involved in the two-phase commit, together with their last known crash counts. At each manager, if this list contains a guardian with a crash count that is greater than the corresponding crash count in the map maintained by the manager, then all currently executing actions that have visited that guardian (or

---

[9] To see why they must be aborted, note that if the handler call requests for those actions had been delayed in the network until after the current request, they would not be allowed to execute.

whose ancestors have visited that guardian) must be aborted. Each manager then merges this new crash count information into its map.

One side-effect of the addition of crash count maps is that the size of the abort lists used in detecting orphans can be reduced if desired. Whenever the manager for guardian X learns of a new (higher) crash count for some guardian Y, it can remove from its abort lists any actions that were created at Y or that have an ancestor that was created at Y. Further, when the manager for guardian X writes its top-level abort list and crash count map to stable storage, it need not write out any actions that were created at X or that have an ancestor that was created at X, since the guardian will restart after a crash with a higher crash count.

References

1. Lampson, B. and Sturgis, H. "Crash Recovery in a Distributed Data Storage System," Xerox PARC, Palo Alto, CA, April 1979.

2. Moss, J. E. B. "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

3. Eswaren; K. P., Gray, J. N, Lorie, R. A., and Traiger, I. L. "The Notion of Consistency and Predicate Locks in a Database System," Communications ACM 19, 11 (November 1976), 624-633.

4. Liskov, B. and Zilles, S. N. "Programming with Abstract Data Types," Proceedings ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), 50-59.

5. Gray, J. N., Lorie, R. A., Putzolu, G. F., and Traiger, I. L. "Granularity of Locks and Degrees of Consistency in a Shared Data Base," In Modeling in Data Base Management Systems, Nijssen, G. M. (Ed.), North Holland Publishing, 1976.

6. Gray, J. N. "Notes on Data Base Operating Systems," Lecture Notes in Computer Science 60, Goos and Hartmanis (Eds.), Springer-Verlag, Berlin, 1978, 393-481.

7. Lamport, L. "Towards a Theory of Correctness for Multi-User Data Base Systems," Report CA-7610-0712, Massachusetts Computer Associates, Wakefield, MA, October 1976.

8. Davies, C. T. "Data Processing Spheres of Control," IBM Systems Journal 17, 2, (1978) 179-198.

9. Reed, D. P. "Naming and Synchronization in a Decentralized Computer System," MIT/LCS/TR-205, MIT Laboratory for Computer Science, Cambridge, MA, October 1978.

10. Liskov, B. "On Linguistic Support for Distributed Programs," Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, PA, July 1981, 53-60.

11. Liskov, B., Snyder, A., Atkinson, R. R., and Schaffert, J. C. "Abstraction

Mechanisms in CLU," Communications ACM 20, 8 (August 1977), 564-576.

12. Liskov, B. et al., "CLU Reference Manual," Lecture Notes in Computer Science 114, Goos and Hartmanis (Eds.), Springer-Verlag, Berlin, 1981.

13. Herlihy, M. and Liskov, B. "A Value Transmission Method for Abstract Data Types," Computation Structures Group Memo 200-1, MIT Laboratory for Computer Science, Cambridge, MA, July 1981.

14. Liskov, B. and Snyder, A. "Exception Handling in CLU," IEEE Transactions on Software Engineering 5, 6 (November 1979), 546-558.

Publications

1. Herlihy, M. P. and Liskov, B. H. "Communicating Abstract Values in Messages," Computation Structures Group Memo 200, MIT Laboratory for Computer Science, Cambridge, MA, October 1980.

2. Johnson, P. R. "Index to CLU Reference Manual," Computation Structures Group Memo 204, MIT Laboratory for Computer Science, Cambridge, MA, February 1981.

3. Liskov, B. H. "On Linguistic Support for Distributed Programs," Computation Structures Group Memo 201-1, MIT Laboratory for Computer Science, Cambridge, MA, June 1981.

4. Liskov, B., Atkinson, R., Bloom, T., Moss, J. E., Schaffert, J. C., Scheifler, R., and Snyder, A., "CLU Reference Manual," Lecture Notes in Computer Science 114, Goos, G. and Hartmanis, J. (Eds.), Springer-Verlag, 1981.

Theses Completed

1. Humphries, T. O. "Overloading in Programming Languages with Data Abstractions," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

2. Moss, J. E. B. "Distributed Program Environment," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

3. Tong, L. S. "A Generalized Command Parser," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

Theses in Progress

1. Henderson, Cecelia. "Locating Migratory Objects in an Internet," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1982.

2. Schaffert, J. C. "The Specification and Proof of Data Abstractions in Object-Oriented Languages," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

Talks

1. Liskov, B. H. "Linguistic Support for Distributed Computing," Capri, Italy, October 1980.

2. Liskov, B. H. "Modular Program Construction Using Abstractions," Wang Institute, Tyngsboro, MA, April 1981.

3. Liskov, B. H. "Modular Program Construction Using Abstractions," Oxford University, England, May 1981

PROGRAMMING TECHNOLOGY

Academic Staff

A. Vezza, Group Leader

J.C.R. Licklider

Research Staff

M.S. Blank
M. Dornbrook
S.W. Galley
L. Hawkinson
P.D. Lebling
S.S. Pinter
C. Reeve
R. Sangal
M.I. Travers

Graduate Students

P.C. Lim
T.F. Michalek
W.J. Noss
S. Ross
K. Wallace

Undergraduate Students

B. Bauman
S.H. Berez
D. Brackman
M.H. Bulat
S. Furst
A. Ghaznavi
S. Schad
D. Scrimshaw
M. Terpin
F. Tou

Support Staff

J.A. Janoff
J.L. Schoof
D.G. Venckus

Programming Technology

1. INTRODUCTION

The Programming Technology Group has continued its study of the process of planning and its development of a Planning System. Our major accomplishments over the past year have been: (1) the development of an initial version of an Advanced Message System (AMS) which is structured to incorporate multi-media messages, a friendly user interface and some knowledge about the topology of the networks through which it can communicate; (2) The study and development of knowledge-based planning aids to assist high-level and middle-level planners; (3) The development of a machine independent MDL language in which the AMS and knowledge-based planning aids will be written. A version of machine independent MDL currently runs on TOPS-20, and it is expected that by next year it will run on Apollo and other 68000 based microcomputers and on VAX computers.

2. ADVANCED MESSAGE SYSTEMS

The Advanced Message System is intended to facilitate multi-media communications among the members of a community distributed of users and servers. AMS minimizes (or isolates) system dependencies. It exists in a heterogeneous environment containing many different networks, workstations, and host computers. The items exchanged are *dispatches*, which are MDL data items. A dispatch may be a message from one person or program to another or input data item to a service or an output data item from a service.

2.1. Transport of COMSYS to TOPS-20

The message system consists of: COMSYS, a mailer; READER, a mail reader-composer; and POD, a mail composer. It was transferred from the PDP 10 and the ITS operating system to run on TOPS-20 and interface with the TOPS-20 operating system and mail environment.

The entire system now has common MDL sources for the ITS and TOPS-20 operating systems. System dependencies are isolated by moving them into packages which are loaded only on the system that needs them, or into segments of code which are compiled only for the appropriate system.

The first type of system dependency results from qualitative differences in the

operating system environments. For example, on TOPS-20 the mail system must be able to put out **MAIL.TXT** files, which have a rigidly prescribed format. On ITS, on the other hand, it is useful to be able to interpret and transmit request files which were written for the COMSAT mailer demon. It would not be useful to have these capabilities on the wrong machine, so they are isolated into discrete packages which are loaded only if appropriate.

The second type of dependency is simpler. It results from differences between the ITS and TOPS-20 operating systems that are minor as seen from MDL. Most of these differences are in the I/O systems, specifically in the formats for file names. They are isolated by use of the OPSYS macro, which expands to the correct code for the operating system specified. The few remaining differences are handled in a specifications file that is loaded when a program's SAVE file is created.

To make the interactive programs system-independent requires the implementation of a terminal graphics package in MDL. On ITS, it produces ITS virtual terminal codes. On TOPS-20s with VTS (Virtual Terminal System), VTS codes are produced. On other TOPS-20s, these operations are simulated (with the aid of knowledge of particular terminals).

2.2. Interfacing to the TOPS-20 World

The world of TOPS-20 mail is rather different from that of ITS mail. The basic vehicle of mail delivery is the **MAIL.TXT** file. Mail from local users is usually delivered to this file, in text form, by the mail composing program of the sender. In other cases (mail from other sites and fully-protected mail files), it is delivered by the XMAILR demon, which runs as part of the system job and has WHEEL privileges.

In the COMSYS world, all mail goes through the COMSYS demon (so that it may be stored in a central database of mail), and the mail is usually converted to text only at "presentation" time. Until then, it is stored as MDL structures, either in the central data base or in the user's local data base.

In order to enable users of COMSYS/READER to live in the established **MAIL.TXT** world, two modules were added to COMSYS. One module outputs **MAIL.TXT** formatted text messages, so that mail from READER users can go to non-READER users. The second module converts a **MAIL.TXT** file to appropriate data-structures to counterfeit a COMSYS-style message and inserts the messages from that file into the user's local data-base. It then (optionally) deletes the messages from the **MAIL.TXT** file. The header parser already in COMSYS was modified and expanded for this purpose.

2.3. Multi-net Environment

As time goes on, more and more networks are being connected in increasingly complicated topologies. As these networks and the sites on them proliferate, it becomes imperative that delivery to such sites be handled in as general a manner as possible.

The major problem which must be dealt with in the multi-net environment is that of networks which are mutually accessible only through remote sites, called *gateways*. For example, from the point of view of the MIT-DMS machine (which is on the Arpanet), sites on the Chaosnet (such as MIT-EE) are accessible only through MIT-MC, MIT-AI, or MIT-XX, which are Arpanet-Chaosnet gateways.

Using the raw data provided by the HOSTS database of networks, COMSYS constructs a table showing the gateways between various pairs of networks. When delivery to a site on a non-contiguous network must be made, the table is searched for a gateway between the sending network and the receiving network. Potentially several gateways would be involved, as in a transmission between a site on the MIT Chaosnet and the Stanford University SU-net.

To avoid clogging gateway hosts with messages destined for third sites which are temporarily unavailable (the "Smith effect"), a protocol has been created whereby the sender site can get the gateway's view of a third site, and use it as a guide for forwarding. For example, if the Arpanet connection between MIT-DMS and MIT-XX is down, mail may be forwarded via the Chaosnet and the MIT-MC gateway. However, if it is MIT-XX, itself, rather than the connection between it and MIT-DMS, that actually is down, there will be wasted effort because the gateway will not be able to transmit to MIT-XX either. Consequently, the gateway would be asked first if MIT-XX is up on the Chaosnet. The gateway could just report the most recent information it had or try to initiate a Chaosnet connection. In either case, it would return the information to the sending site.

The term "Smith effect" comes from an old story. Two men worked in a large office in the Pentagon. The first worked from ten to four (if that), and always seemed to have free time and an empty in-box. The second spent horrendous twelve-hour days and weekends, and was constantly harried and overworked. Finally, the second fellow couldn't stand it any longer. "How is it," he asked the first, "that you get your work done so easily? You must have some system."

"Sure," the first replied. "Whenever I get something that looks like it's complicated or time-consuming, I just write on it 'To Smith for Action.' In an organization as large as this one, there has to be a Smith."

The second man stared at the first: "I'm Smith!"

2.4. Dispatch Environment

The COMSYS/READER environment is being used as the base on which to build the multi-media message system. As mentioned, the basic currency of this system is the *dispatch*. A dispatch may contain a message, or it may contain something more structured, such as a request for action or the report of some action completed. The dispatch itself is the envelope (and as such, analogous to the objects transmitted by Mail Transfer Protocol [1]). The transmission mechanism does not look "inside the envelope," which may contain any MDL data structures. Dispatches are transmitted to and from *services*, which may include mail-sending, forwarding, archiving, and so on.

A version of COMSYS was brought up which transmits dispatches, and some experiments were performed. A "Finger Server," which takes as input a dispatch asking for the "Finger" data (information about an individual user supplied by the user) on a given person, was implemented. A READER command asking for such data was implemented. It sends a dispatch requesting "Finger" data and then returns to the user, but awaits a reply (in the background). When the reply appears, it is displayed. This rather simple experiment allowed us to define protocols for dispatch delivery, service selection, and asynchronous action requests in the READER.

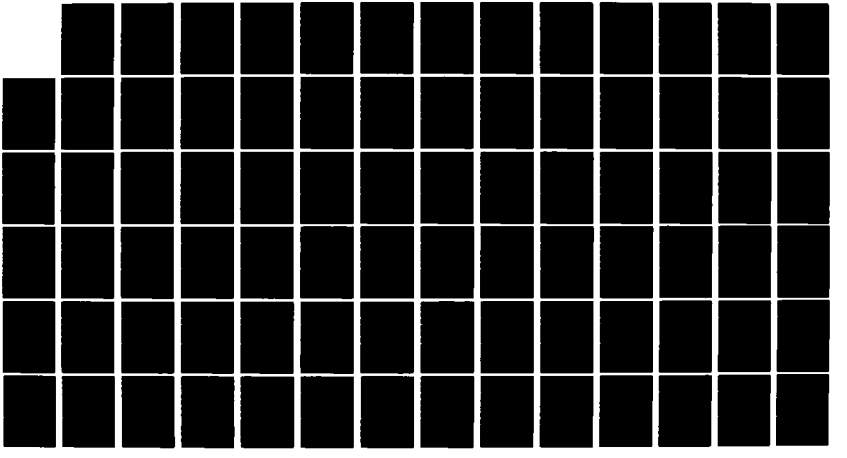
2.5. New User Interface

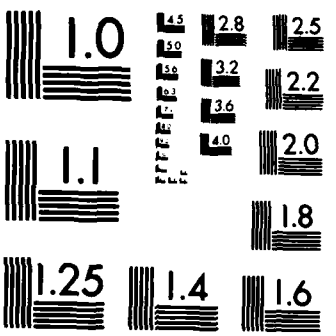
A new user interface was written, intended to be easily adapted for use with a variety of input devices. It is based on the principle that input devices should communicate with the parsing and execution part of the interface in a very stylized way. That is, the software encapsulation of a device puts characters into an input buffer and requests special actions (such as "Execute" or "Help"). All devices are expected to adhere to this convention.

The basic interface is similar to a conventional line editor (though it may be tailored to mimic one's favorite editor). The interface keeps track separately of the cursor and the parsing cursor (or *parsor*). For example, the user may enter an entire line of command, edit it at will, and only then ask for the command to be executed. Until he does, the parser remains at the left end of the line. It moves to the right as the line is parsed, and stops when the line is fully parsed or an error is encountered. In the latter case the cursor may be moved to the point at which the error was detected, and the user may do further editing.

Special actions such as "Help", "Prompt" or "Show possibilities" may be handled specially by particular commands, or they may "fall through" to default handlers. In general, such informational actions are expected to modify the input buffer minimally (by using other words, for example).

AD-A127 586 LABORATORY FOR COMPUTER SCIENCE PROGRESS REPORT 18 JULY 3/3
1980-JUNE 1981(U) MASSACHUSETTS INST OF TECH CAMBRIDGE
LAB FOR COMPUTER SCIENCE M L DERTOUZOS 01 APR 83
UNCLASSIFIED LCS-PR-18 N00014-75-C-0661 F/G 9/2 NL





The user-interface has been implemented initially for use with a keyboard and display screen, and will be upgraded as other input devices become available.

3. KNOWLEDGE-BASED PLANNING AIDS

The primary goal of this project is a system that can assist in planning and management within an organization. Our efforts are focused on high-level (strategic) planning and middle-level planning, rather than on low-level (operational) planning. The system we are building is designed to provide aid to planners, rather than to do planning on its own. This is in recognition of the existing state of the art and the fact that high-level planning involves social and political factors that are too little understood to be amenable, at present, to modeling in a knowledge-based system.

For our planning aid system to assist a user in active, high-level planning, it must provide the following capabilities: record-keeping, history reporting, status reporting, projecting (e.g., future expenditures), document generation (e.g., of proposals, budgets, and schedules), and alerting. Each of these capabilities is useful in its own right, apart from any particular planning effort.

Though our goals are ultimately pragmatic, our work thus far has been primarily theoretical, and mostly within the realm of AI. We have developed a new knowledge representation formalism, called "PREP". PREP deals specifically with the issues of time and hypotheticality that are so central to planning. PREP may be thought of as a specialized logic featuring, among other things, a sublanguage for expressing predicates, a quotation operator, and a formal semantics. (1) The sublanguage for expressing predicates is essentially a calculus of binary relations operating upon interpretations of words taken from English and other languages. (2) The quotation operator is similar to the quotation operators of LISP and Montague grammar. (3) The formal semantics is more than denotational, in that expressions may be assigned interrelated interpretations at several distinct levels of abstraction. The levels of abstraction are the intentional level (the least abstract), the relational level (where time dependencies are explicit), the extensional level (i.e., denotational), and the propositional level (the most abstract). The formal semantics of PREP, as well as the model of planning discussed below, rest upon a general model of the world that deals with time and hypothetical entities. PREP has been implemented in MDL as a general-purpose knowledge-base system.

3.1. A Model for Planning

The model of planning we have developed is more flexible and well-suited to our needs than the "backtracking" models that have been successfully used in work on such well structured and highly constrained problems as route planning. Our model permits unstructured, incremental elaboration (design) of the hypothetical or actual entity being planned, as well as concurrent exploration and comparison of alternative possibilities. Instead of working at any one time within a single current context or world state (as required by most AI models of planning), we work at all times within a unique world that may contain incompatible hypothetical entities. Each hypothetical entity in this world is described by a particular set of hypotheses (i.e., a theory) expressed in a knowledge representation formalism like PREP. A hypothetical entity is "possible" if the set of hypotheses describing it is consistent, not only per se, but also with all applicable constraints.

The key structure used in our planning model is an "elaboration tree". Each node *e* of an elaboration tree focuses on (pertains to) a hypothetical entity. The hypothetical entity is described by the set of hypotheses consisting of all hypotheses at *e* or at any ancestor of *e*. An elaboration (node in an elaboration tree) may bear properties, such as the incompatibility of its associated hypotheses with those of other specified elaborations, or the impossibility or inferiority of the entity it focuses on.

The process of planning may be described in terms of the following sorts of operations on elaboration trees: spawning a new elaboration; adding detail to an elaboration, either as an hypothesis or as a derived consequence; testing an elaboration for consistency; comparing two elaborations; rejecting an elaboration because of inconsistency or inferiority; combining two compatible elaborations; eliminating elaborations that are no longer useful; choosing among alternative elaborations; and "accepting" an elaboration by treating its associated hypotheses as facts or expectations.

Another important part of our model of planning, besides the elaboration tree and operations upon it, is the process of "viewing". In viewing--constructing and displaying a view--certain (derivable) entities are derived (from other entities). In terms of computation, viewing consists of filtering, then evaluation, and finally "presentation".

We are representing general and domain-specific knowledge in PREP in a way that is strongly guided by our world model. The general knowledge consists of descriptions of commonly used entities and actions. It is not limited in its use to our domain.

As for domain-specific knowledge, we have represented knowledge about an

organization's goals and objectives, its department and program structures, relationships between programs and departments, flow of data and control budgets and proposed expenditures, business transactions, contracts, etc.

We have also represented knowledge specific to a particular organization we are trying to model. Our focus is on its budget-related activities. We have modeled the specifics of its program and department structures, its goals and objectives, the structure of its budget, and actions which are carried out in preparing its budget.

The task of building the knowledge base is continuing steadily at all three of these levels.

3.2. Other Knowledge-Base Planning Aid Projects

A version of the LISP LOOP iteration facility for the MDL Programming environment has been implemented. LOOP is a macro that is characterized by a stylized-English form and the ordered execution of clauses. It includes a facility for definition, by the user, of generators that permit the implementation of efficient iteration operators on abstract data. This facility is important for the use of LOOP in the knowledge base system, PREP.

A memory subsystem for PREP is in progress. It will be a library of functions and macros with which users or their programs can modify and/or monitor the state of the knowledge base. An effort has been made in the design of this subsystem to encapsulate implementation detail in such a way as to facilitate modification and transportation.

An MDL program that performs a crude translation from the language of PREP to English has been developed.

A thesis project that will involve the development of a "world model" that will draw from previous work in artificial intelligence, philosophy, and formal logic is planned. The program is intended to be able to reason about action and change. It is intended to evaluate plans in such a way as to determine whether they will indeed achieve their goals without violating any of their static or dynamic constraints.

An extension and formalization of the basic logic of PREP and on strategies of inference within PREP is in progress. Work on the proof of theorems in PREP and on a MDL program that does simple inheritance reasoning with a few PREP inference rules and theorems has been developed. In addition, a study of formal logical systems, including Propositional Logic, First Order Logic, Modal Logic, Tense Logic, and Non-monotonic Logic has been done. A master's thesis project on the design of a formal logic for a planning system that will deal readily with

hypotheticals, the branching of future possibilities, time, and free quantification is planned. This thesis project will feed into the evolution of the PREP formal logic.

4. MDL

The implementation of the Planning Aid System and the Advanced Message System (AMS) will be in the MDL language, which has been the Programming Technology Group's primary development language for the past nine years. The MDL used will be a new machine-independent implementation. The reason for this choice of language is twofold:

1) Members of the Laboratory have extensive experience with MDL and have used it successfully in the past for the design and development of large systems, including message and database systems. Some portion of the AMS will be extracted from code already being used for the DM machine's communications system (COMSYS).

2) Since the AMS will be composed of modules (e.g. servers, file stores) distributed over a number of machines which may include various processors, there is a great advantage in writing the system in such a way that any module can be transferred among the machines with a minimum of effort.

Our goal, therefore, has been to implement this machine-independent MDL, which will be generally compatible with the current MDL language, and which will perform well on the proposed target machines (DECSYSTEM-20, Apollo, and VAX). The approach to this goal has been to develop a virtual machine running a language called MIM (Machine Independent MDL). This approach is analogous to that taken in the implementation of PASCAL.

The project is divided into five major subprojects, which are outlined below.

1) Design of the virtual machine (called MIMI, for MIM Interpreter). This design was completed last year and is detailed in the MIM Design Document, SYS18.01.

2) Implementation of virtual machine interpreters for the various target machines. At the present time, a DECSYSTEM-20 interpreter is operational and an Apollo interpreter is being designed and implemented.

3) The MDL Compiler (MIMC) which translates MDL code into the machine-independent format. A working MIMC now exists.

4) The MDL interpreter (called MUM) written in MDL itself. A MUM is now operational.

5) Various MIM Order-Code compilers (MIMOCs) for the various target machines. The MIMOCs will take the output of the compiler (MIMC) and produce executable machine code for a specific target machine. A MIMOC for DECSYSTEM-20 now exists and one is being designed and implemented for the Apollo.

The important thing to note is that only one thing, the MIM interpreter, need be written in code that is specific to a target machine. All of the other modules (MUM, MIMC, and MIMOC) can be and have been written exclusively in MDL.

4.1. MIMI

MIMI20 is an interpreter for the MIM virtual machine on DECSYSTEM-20 under TOPS-20. It also provides low-level support for MIM code open compiled into TOPS-20 order code. MIMI20 is about six thousand words of TOPS-20 code. It is designed to be small and simple. It is the only part of the new machine independent MDL system that will not be written in MDL.

One problem encountered in bringing up MIMI20 was a problem in bootstrapping. We did not want to build MDL's READ and EVAL into MIMI since they were both being written in MDL as part of MUM (see below). We were faced with the problem of how to read in the MDL programs that are required to read in other MDL programs. The solution to this problem was to build into MIMI a very simple READ and a very simple EVAL that know how to read and evaluate the minimal set of MDL objects needed to build MSUBRs (MDL compiled functions). This very simple program reads in a bootstrap loader. The bootstrap loader includes a READ and EVAL that are more complete than the ones in MIMI but still not as general as the full-fledged READ and EVAL. The bootstrap loader also defines simplified versions of other parts of the interpreter required to load the real interpreter. Once the bootstrap loader has run and loaded the entire interpreter, it is thrown away, along with the part of MIMI used to read it in.

The additional step of going through a bootstrap loader was taken to minimize the amount of code required in each MIMI for bootstrapping. Most of the bootstrapping code is written in MDL and is therefore easily transportable.

MIMI20 is capable of running in extended addressing mode on TOPS-20 so that it can have a virtual address space 23 bits wide. A runtime switch is provided to select whether or not to use extended addressing mode. This was done because extended

addressing on TOPS-20 has been observed to be significantly slower than "normal" addressing. However, extended addressing does give MDL some breathing room in terms of virtual address space.

On April 30, the group acquired an Apollo Domain computer. The Apollo Domain is a personal computer based on the Motorola 68000 microprocessor. Work has begun on designing a MIM for the Domain. The MIMI implementation will be competed by the fall of this year. Since the Domain is a byte oriented machine and MIM is a byte oriented virtual machine, the implementation of MIMI should proceed rapidly.

4.2. MIMC

The MDL compiler has been modified to produce MIM instructions that can be either interpreted by a MIM interpreter or open compiled for a particular target machine. MIMC is written entirely in MDL. It currently runs in "old" TOPS-20 MDL and will soon be transported to run in the new MDL.

Fortunately, a large amount of the work done on the original MDL compiler was directly usable in MIMC. Specifically, the flow analysis code and the type analysis code were used with almost no changes. This code constitutes a large fraction of the code in the compiler. In some of the changes that were made, we were able to simplify the code generators because a given piece of MDL code cannot compile into as many different sequences of output code in MIM as it can when compiling directly into TOPS-20 order code and because register allocation is no longer performed in MIMC. MIMC needs to deal only with named temporaries and the stack. The difficult problems of actual machine code generation and register allocation have been deferred to the MIMOCs for different target machines.

There is an obvious risk of losing performance by taking this approach to compilation. The high-level information known to MIMC can be useful to the order-code compilers. As a partial solution to that problem, MIMC occasionally outputs extra information that a MIMOC may find helpful in generating efficient code. There are currently four examples of this kind of information:

1) Dead temporaries are indicated. This tells MIMOC that it need not store the current value of a temporary back into memory.

2) Record type information can be indicated in references to records. Since the MIM instructions for accessing records require access to a table of record information, they cannot be efficiently open compiled without additional information. When MIMC knows the specific record type, it adds that information to the MIM instruction.

3) Situations in which a zero pointer means false and is used only in a conditional test are indicated. This prevents MIMOC from generating an object of type FALSE, looking at it and then throwing it away. Instead, it can simply branch if the slot is zero.

4) Finally, labels that are at the beginnings of loops are indicated. This can help MIMOC do some register optimization by aggregating the states of the world at the labels.

Other changes had to be made to MIMC to enable it to handle new constructs that did not exist in old MDL. New mechanisms include the new form of DECL, called ADECL, and the mechanism for generating MIM instructions directly from MDL code during compilation.

4.3. MUM

In order to bring existing MDL software quickly into the new machine-independent MDL environment, the MDL interpreter itself has been written in MDL. This brings the added advantage of easier modification and maintenance of the interpreter. (The "old" MDL interpreter was written directly in assembly language.) Preliminary measurements indicate that MUM is at present between one-half and one-third as fast as the "old" MDL interpreter. The speed is expected to improve with optimizations to the MIMC and MIMOC subsystems.

MUM is almost entirely standard MDL code, with one exception: MUM is allowed to perform MIM machine instructions (e.g., creation of bindings, treating ATOMs as structures). This capability is enabled only when the interpreter is compiled.

The MDL language itself is mostly unchanged from the "old" MDL. The ability to "start from scratch" in this project has enabled some alterations to be made and some new capabilities to be added. One major alteration is a new type-declaration syntax which is much easier to read and interpret. New capabilities include multiple-return features and the ability to create TUPLEs within functions. A new I/O system is in the works, and a more straightforward software interrupt system is nearly complete.

Since the MDL interpreter is written in MDL, it is now possible to have a runtime system with only specific modules of the interpreter loaded, thus the new MUM interpreter is usually considerably smaller than the old assembly-language one. In addition, the fact that SUBRs are now identical to 'user code' allows the user to obtain information about the calling sequence for SUBRs within the MDL environment (i.e., by examining the type declaration of the SUBR).

4.4. MDL Environment

There is a rather substantial amount of environment that has become associated with MDL, including a powerful array of debugging tools. Since the changes to the MDL interpreter are relatively few, the transfer of these tools to form the new MDL environment is a high priority for the group in the coming year. In addition to the tools presently available, there are plans to translate into MDL other utility programs which are currently written in assembly language. These include source comparison programs and text editors. The obvious advantage is that an entire programming environment will become available on any system which will run a MIM interpreter, greatly decreasing the time required to make a new piece of hardware useful to the programming community.

4.5. MIMOC

An order-code compiler for the TOPS-20 system (MIMOC-20) has been written in MDL. It takes the output of the MIMC process and generates PDP-10 instructions to execute the various MIM instructions. The great majority of MIM instructions 'open-compile' (i.e. can be translated in-line into PDP-10 instructions). The others (e.g., building structures, calling other pieces of code, etc.) call entry points built into the TOPS-20 MIMI for that purpose. Conventions have been established between MIMI-20 and MIMOC-20 as to calling of subroutines, where values are returned, etc.

Although the currently implemented MIMOC-20 produces quite good code, various optimizations are in the works. A major optimization is in reducing the large overhead associated with subroutine calling. There are two proposed ways of doing this: one is similar to the GLUEing system of the "old" MDL, in which subroutines can simply PUSHJ to each other without the overhead of creating stack frames. The other is simply a pared-down version of the stack frame calling system. In this second system, subroutines which follow various conventions can use a simplified call/return sequence. Both of these optimizations are being implemented at the present time, and both should be operational by this summer. Similar plans are being designed for MIMOC-Apollo which will be implemented this summer. The expectation is that with these optimizations, compiled MDL code will run comparably with code produced by the "old" MDL, which has been found to yield extremely efficient code.

4.6. Machine-Independent MDL Graphics

A machine-independent MDL graphics system was designed. The machine-independent MDL now under development will run on a variety of machines, many with bit-map displays and diverse input devices. The MDL graphics system,

DIGRAM, is designed to support a virtual display and a variety of characteristic input devices and to facilitate transport of MDL programs.

DIGRAM uses the same virtual-machine interpreter scheme as MDL itself, including the concept of compilation into a particular target machine's order code to gain execution speed.

The basic elements of a DIGRAM display are display segments. Segments are mapped into a virtual display-space, the *world*. Areas of the world are in turn mapped onto the real display through *viewports*, which perform appropriate clipping and scaling.

Segments are built up out of triangles (and so can be used to construct arbitrary polygons), lines, markers (used as cursors), and text. Triangles and lines may be scaled and rotated.

DIGRAM defines several types of abstract input devices. These are *picks* (selectional devices), *valuators* (one-dimensional values), *locators* (two-dimensional values), *keyboards*, and *buttons*. Most real devices are neatly abstracted to one of these or composites of several of them. For example, a mouse is usually a locator plus buttons and can be used as a pick. DIGRAM permits interrupts to be enabled for asynchronous devices such as buttons.

Design and testing of display algorithm are being carried out on the TOPS-20 machine with simulated output on a VT100 display.

4.7. Machine-Independent Data Transmission

A machine-independent MDL data transmission protocol was designed and implemented. The MDLs on various machines will have nearly identical appearance to their users, but may be very different internally, as target machine implementations may vary considerably. Nevertheless, users will transmit data between these different implementations. Such transmission could be done using READ and PRINT, but unfortunately, transmissions by that mechanism do not preserve sharing and circularity.

Previously, both MDLs had the same internal implementation (that is, the representation of a data-structure in the machine was identical whether the MDL ran on ITS or TOPS-20). It was therefore possible to use the MDL GC-DUMP and GC-READ SUBRs to transmit data in internal format, preserving sharing and circularity.

In this new design, many of the features of GC-DUMP and GC-READ are preserved, but the MDL structures are represented in a byte-encoding which is

implementation-independent. This implementation still requires a "mini-GC" to produce the structures to be transmitted. It is best used on large MDL structures, where the GC-time is a smaller percentage of the transmission time.

References

1. Postel, J. "Internet Message Protocol," University of Southern California Information Sciences Institute Report RFC 759, Marina del Rey, CA, August 1980.

Publications

1. Dornbrook, M. and Blank, M. "The MDL Programming Language Primer (Draft)," MIT Laboratory for Computer Science, Cambridge, MA January 1981.

2. Golden, V. E. and Schoof, J.L. "Introduction to MIT-XX," MIT Laboratory for Computer Science, Cambridge, MA, March 1981.

Theses Completed

1. Bulat, M.H. "Interactive Maintenance Terminal Fault Isolation Program," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1981.

2. Goldman, C.E. "An Interactive Modeling and Simulation System for Structured Engineering Design," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 1981.

3. Harris, L.A. "An Estimating Method for Automatic Screw Machine Products," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

4. Holt, D.A. "Computer-Aided Task Scheduling," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

5. Noss, W.J. "Database Alerting Mechanisms," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA June 1981.

Theses in Progress

1. Lim, P.C. "A Virtual Device-Independent Graphics System for MDL," S.M. thesis, MIT Department of Electrical E..gineering and Computer Science, Cambridge, MA, expected December 1981.

REAL TIME SYSTEMS

Academic Staff

S. A. Ward, Group Leader
M. L. Dertouzos
R. H. Halstead
R. Zippel

Research Staff

C. Baker
J. Gula
S. Keohan
R. McLellan
J. Test

Graduate Students

T. Anderson
D. Goddeau
M. Johnson
A. Mok
J. Powell
L. Seiler
J. Sieber
R. Simmons
T. Sterling
T. Teixeira
C. Terman
J. Troisi

Undergraduate Students

J. Arnold
C. Eliot

Support Staff

O. Feingold
M. Goliaszewski
R. Kane
L. Kenen
E. Tervo
P. Vancini

Real Time Systems

1. INTRODUCTION

Major research activities of the Real Time Systems Group during the past year have been (1) continued development and technology transfer of the Nu personal computer; (2) design and initial implementation of the TRIX operating system; (3) continued investigation of the MuNet and similar scalable multiprocessor architectures; and (4) continued research in the area of VLSI design tools.

2. NU: THE LCS PERSONAL COMPUTER

The dissolution of the manufacturing with Zenith in the Fall of 1980 has led to a fairly serious perturbation of the plans and schedule for Nu production. First-order effects of this development were

- The expenditure of considerable effort in the search for a new industrial connection, negotiation of a contract, and transfer of the Nu technology. This effort has resulted in our current contract with Western Digital, under terms similar to those we had with Zenith, for their long-term manufacturing rights to the Nu.

- An undeniable setback in the schedule for availability of the Nu to the Laboratory; we are currently aiming for first deliveries from Western Digital (aside from a possible evaluation prototype) in first quarter 1982.

- The re-opening of a variety of technical and design issues.

The reconsideration of aspects of the Nu implementation may be viewed as a mixed blessing. On one hand, it involves something of a backwards step in the progress of the Nu design; on the other hand, it affords an opportunity to correct a variety of weaknesses in the previous implementation.

Planned changes in the Nu design pursuant to production by Western Digital include:

- Packaging. Western Digital engineers favor a smaller number of somewhat larger boards (on the order of 200 square inches), in the interest of lower costs. This change affects primarily the increments by which a system can be expanded; a memory board, for example, will contain at least 1 MByte.

- Functional improvements. A cooperative effort between Western Digital and the Nu team (Arnold, Goddeau, Gula, McLellan, Terman, Ward) has led to plans for a high-speed cache. These changes are expected to substantially improve Nu performance by providing the higher effective memory bandwidth consistent with current (8 MHz) and future processor chips.

- Indulgence of professional idiosyncrasies. A variety of changes, particularly to the NuBus, reflect fundamentally "religious" differences between the Western Digital engineers and us. Typically these are rooted in the greater familiarity of the engineers with alternative technologies. We recognize the importance of catering to the areas of expertise of the Western Digital designers, and attach high priority to their confidence in the technical aspects of the system they are to manufacture. While most such technical details are as yet unresolved, we have taken the position that, so long as serious functional or architectural compromises are not involved, the technological idiosyncrasies of the manufacturer should take precedence over our own.

Our working relationship with Western Digital has been very encouraging. There is a mutual respect between the technical teams at MIT and WD, and consequent cooperation in design decisions. Jim Gula has left the LCS staff to join WD; while we regret losing him, we feel that he will serve important roles both in the MIT/WD interface and in making the Nu into a successful product.

A variety of embellishments and functional extensions to the Nu have been developed during the past year. These include the PSI, a satellite processor with floating-point capabilities developed by Arnold. The PSI features an 8086/8087 processor pair and local RAM, and may be used both to provide effective floating-point processing to user programs running in the main processor and to support dedicated functions.

A Canon laser printer has been interfaced to the NuBus, using existing video boards and a substantial driver program to perform the scan conversion. The output from the Canon printer falls just short of Dover quality but is substantially better than that from the XGP or Varian/Versatec devices, and is particularly appealing because of the modest size (that of a compact office copier) and cost (7000 dollar range) of the unit. Similar devices are becoming available from other manufacturers; they constitute a potential means to distribute the service currently provided by the Dover, improving user accessibility and reliability.

Other Nu developments include a video lookup table for color and gray scale and a 64-voice audio synthesizer.

3. TRIX AND UNIX IMPLEMENTATIONS ON NU

The initial TRIX system, including UNIX-compatible user-program interface, was completed in December 1980. While this system includes all of the functionality that had been planned for the first implementation, it lacks various embellishments (e.g., system processes) devoted to performance rather than function; consequently, it falls short of an acceptable performance level for routine use in other applications. In December, it was concluded that the research interests of the TRIX project were best served by a re-engineering of its internal structure rather than by polishing of the existing implementation. In order to reconcile this plan with the need for a viable Nu programming environment for other projects, Version 7 UNIX was ported to the Nu during the Spring by Gula, Sieber, Terman, and Test. At this point substantially all major UNIX components have been ported to the Nu, and a stand-alone Nu may be used to maintain the system itself.

A modest amount of planning has been devoted to the new TRIX implementation by Halstead, Sieber, and Ward; detailed design and implementation will begin during the summer. A general goal of the restructuring is to capture the efficiencies of conventional (UNIX-like) mechanism when dealing with synchronous, procedure-call-oriented programs, rather than encumbering every interprocess communication with the burdens associated with asynchronous message-passing. The new structure makes a clear separation between the static environment of a process (including memory map, open files, etc.) and its thread of control (*viz.*, its stack), allowing *e.g.* control to follow a message from one domain to another without scheduler intervention and overhead. Asynchronous communications, *e.g.* having several outstanding messages simultaneously, involves extra overhead; but the implementation will default to conventional mechanism in the common synchronous situation, much as a spaghetti stack operates efficiently so long as no retention is required.

4. OBJECT-ORIENTED MULTIPROCESSING: THE MUNET

The MuNet research project investigates the construction of highly concurrent machines that support a LISP-inspired, object-oriented model of computation. A word coined to describe the kind of machine envisioned is "*myriaprocessor*" --a computing system constructed from a myriad of individual *nodes*, each with its own processing and memory capabilities [1]. Desirable features of a myriaprocessor architecture would be (1) scalability, (2) communication cost reduction through locality, and (3) flexibility.

Webster's Dictionary defines the prefix "myria" as denoting "(1) many, numerous... (2) ten thousand" [2]. Both senses of the prefix suit the present context,

symbolizing our interest in computing systems composed not merely of a few or a few dozen computing nodes, as one often envisions a system called by the name "multiprocessor," but of a myriad of nodes, like ants in an anthill, each contributing its small part to the functioning of the whole. Such a system, if it can be made to work, has several attractions:

1) An anthill can operate in more or less the same fashion over a considerable range in the number of ants. Similarly, it should be feasible to build computing systems following an architectural philosophy *scalable* over a wide range in the number of nodes. Such a system could be upgraded by the simple addition of more nodes, and would undergo graceful degradation in performance as nodes were removed.

2) An increasingly apparent feature of integrated circuit technology is the disparity between the speed of operations local to one chip (or even one neighborhood of a chip) and the speed of off-chip (or even across-chip) communications. Present-day computer architectures emphasize specialized components and high-bandwidth communication. A future architecture composed of general-purpose "ants" each implemented as a single chip, or even just a portion of a chip, could avoid global communication for strictly local computations, and match the technological constraints much better.

3) Such systems could be built almost entirely out of a large number of one standard component: a single-chip node. Their capacity would be adjusted not by redesign, as in today's families of compatible processors, but simply by adding or subtracting nodes.

In order to approach the capabilities desired in tomorrow's computers, a myriaprocessor must use many more nodes, more closely cooperating, than multiprocessors composed of larger elements. It is thus that both senses of "myria" apply: in emphasizing a qualitative difference from multiprocessor architectures, and in giving a numerical estimate (10,000) that hints at the sizes up to which one would hope to find this philosophy applicable.

For such a vision to materialize, several obstacles must be surmounted. The capabilities required in an individual node must be scaled down to the point where single-chip implementation (including the requisite memory) is realistic. Programmers, programming languages, compilers, and interpreters must become able to cope with the concurrency that a myriaprocessor requires in order to be effective. Strategies for effectively distributing a workload among the resources of a myriaprocessor, and avoiding thrashing, must be developed.

4.1. Implementation Outline

Several paradigms could be used for the architecture of myriaprocessors. Data flow graphs are being studied actively by many researchers [3] [4]. Others have suggested actor systems based on message-passing [5]. Similar diversity is evident among implementation mechanisms proposed to support these models of computation. Ideally, such mechanisms should meet the following criteria:

1) They should exhibit a considerable degree of locality. Architectures in which communication between any pair of nodes is equally easy are condemned to have a cost that rises faster than linearly with the number of nodes, compromising scalability.

2) Their model of computation must be amenable to some reasonable software engineering discipline. The model of computation chosen must aid the programmer in coping with the complexity of his task, while at the same time providing sufficient power to implement the desired system and making possible sufficient concurrency of execution.

3) The behavior required of individual nodes should be as uncomplicated and undemanding (with respect to space and time requirements) as possible, to allow maximum flexibility of implementation and minimize the size of each node. It is desirable to keep open as much as possible of the implementation spectrum toward smaller and more numerous nodes.

Model of Data: The object-oriented model of computation of, e.g., LISP [6], contrasts with the more value-oriented semantics of the data flow model. We are more concerned with arguing for the viability of an object-oriented approach than for its superiority; nevertheless, we note in passing that the object-oriented approach is closely associated with a variety of languages in production use, in contrast to the more radical value-oriented approach. Facets of the latter, such as its incompatibility with side effects on data structures, are beneficial for parallel implementations, but pose challenges to the programmer that several creative additions to the value-oriented languages (such as Id [3] and VAL [4]) have only partially resolved.

For current experimental purposes, only one kind of object-- the familiar LISP *conscell*, with two components called **car** and **cdr**, each of which can contain a primitive datum or a reference to another conscell-- is supported. It is of course possible to include other kinds of objects, such as vectors, arrays, character strings, *etc.*, as most LISP implementations do.

The reader should bear in mind that the notion of conscells is only an abstraction; allocation of two machine words per conscell, each containing a pointer, is only one implementation possibility. More space-efficient representations, such as cdr-coding [7] [8], have been proposed and used.

Implementation of the LISP Model: In a myriaprocessor, portions of the data in the system (*i.e.*,conscells) would normally reside at various different nodes. In the most general model, an individual conscell (*i.e.*, information about its contents) may reside on one or more of the nodes in the system, and may be referenced from conscells stored on yet other nodes. The system may be called upon to deliver up the contents of a conscell, given a reference to the conscell, update the contents to a new value, or create a new conscell with specified contents. Furthermore, unreferenceable conscells must be garbage-collected and recycled into the allocation process. *Reference trees* [9] [10], among other schemes, support all of these functions, for arbitrary distributions of objects across nodes.

Beyond the bookkeeping required merely to maintain the database of objects in a constant state, it may be desirable or necessary to move copies of objects from one node to another, or create or delete duplicate copies of objects. Decisions about such operations involve strategy considerations of just how the processing and memory load is to be shared by the nodes.

Interpretation Mechanism: Most implementations of LISP represent a program as a recursive data structure, effectively a parse tree, constructed out of conscells. An *interpreter* algorithm walks this tree, manipulating other data structures such as a stack and environment, that can also be built out of conscells. Direct hardware implementation of such an algorithm has recently received some attention, notably in the SCHEME single-chip microprocessor [11].

For the present purposes of the MuNet, however, a language (dubbed LCODE) has been chosen that is more like the stack-machine intermediate code used by many compilers. Programs are still represented as LISP data structures, but as considerably "compiled" and rearranged versions of the higher-level originals. This representation is better to highlight the machine-level details that we consider, and emphasize that the language in which programs are written need not be LISP (indeed, LISP *per se* does not deal with the issue of concurrency).

"Operation codes" for programs in our system are special atomic symbols such as **cons**, **car**, **plus**, **if**, *etc.*; a program is a LISP data structure including references to these atoms. As with more traditional LISP, our interpreter uses a stack (for control information) and environment (for variable bindings); the various primitive operations are defined in terms of their effect on these. For example, execution of a **cons** operation pops two items (object references) off the stack and pushes a reference to a newly created conscell with the two popped items as its contents.

An interpreter for our simple stack code can be described in terms of three registers, *PC*, *STACK*, and *ENV*, each containing a reference to a LISP object. The stack can be maintained as a LISP list. Parallelism can easily be incorporated into this low-level model by allowing for several interpreters, each with its own private *PC*, *STACK*, and *ENV*, to coexist and execute concurrently. In operation, the system thus resembles a large web of inter-referencing conscells, with a multitude of "spiders" (interpreter tasks) crawling along the filaments of this web, extending and changing it.

Interpreter Implementation: In a multiprocessor system with objects distributed among different nodes, it will generally be impossible to find one node containing all the information needed by a particular task. Either the task must move from one node to another as it references different objects from its *PC*, *STACK*, and *ENV*, or copies of objects must be moved to make available the requisite information. This choice is basically a strategy decision and affects the distribution of processing and memory load among the nodes in the system. The two extreme alternatives along this dimension are (1) never move tasks, always move objects, or (2) never move objects, but always move tasks to the location of the needed information. Adopting a position between these extremes involves building hardware for moving both tasks and objects, so some savings would be possible should either extreme prove viable.

Alternative (1) is not particularly attractive; taken literally, it would force all descendants of a given task to occupy the same node, and prevent any real concurrency among them. We would like to be able to redistribute such tasks about the system. Alternative (2) is more appealing. It is at least plausible that by suitably distributing objects among the nodes (recall that the availability of frequently accessed objects can be enhanced by making multiple copies of them), and choosing wisely where to create new objects, the natural flow of tasks from node to node as they reference one object and then another could lead to good usage of system resources and avoid excessive communication overhead.

Probably even alternative (2) is not sufficient as the only object and task distribution mechanism on the system; however, it *is* attractive as a short-term mechanism that can be used from instruction to instruction by the hardware of individual nodes. Responsibility, and mechanism,, for longer-term distribution could then reside in special "introspection" tasks, which would observe the flow of system operations and effect appropriate adjustments.

Synchronization and Side Effects: The key feature of a myriaprocessor is its ability to effectively support a large number of concurrent activities, achieving high performance through large-scale parallelism rather than very-high-speed individual components. To make possible the decomposition of a program into parallel tasks,

the system must provide suitable facilities for communication and synchronization between tasks. Tasks can interact, in our LISP model, only via changes in the shared web of conscells; *viz.*, by side effects (updating the contents of an already existing conscell to refer to different objects). Thus synchronization among tasks must be effected by a suitable pattern of such side effects.

Many different synchronization problems and mechanisms have been explored in the literature, but the capabilities required can be put into two categories: the ability to enforce *precedence constraints* between operations in different tasks, and the ability to enforce *mutual exclusion* between tasks sharing a resource. If these low-level capabilities are present, higher-level synchronization constructs can be synthesized.

The need to enforce precedence constraints between tasks often appears in a "fork/join" situation, *e.g.*, when it is desired to evaluate in parallel several operands of a **plus** operator, then collect the results and produce their sum. If an activity *A* is to precede ant activity *B*, this precedence constraint can be enforced by means of a conscell shared between *A* and *B*. *B* can busywait looking for a value to be placed in the conscell upon completion of *A*. This solution is unaesthetic, however, and could cause substantial overhead in the presence of large numbers of waiting tasks.

Mutual exclusion between tasks can also be performed by busywaiting if an atomic test-and-set operation is available, but, once again, this solution is inelegant and potentially inefficient. To implement a mutual exclusion construct that is fair and/or avoids busywaiting, more mechanism is needed. Such mechanism often involves instructions to read and write several values (e.g., splice a task into a queue of waiting tasks) in one atomic operation. If all the memory locations (in our case, **cars** and **cars** of conscells) involved in the atomic operation are located at one node, such an atomic operation can be provided fairly simply by some equivalent of the expedient of temporarily disabling interrupts on a conventional machine. It is much more complicated, though, to implement atomic operations involving objects stored in different nodes. We prefer to avoid both this complication and the alternative complication and the alternative complication of having to move the relevant objects all to one node, so we would rather find *single-object* atomic operations that suffice to implement interesting synchronization constructs in a desirable manner (*e.g.*, without busywaiting). A simple-object atomic operation has the property that all the memory locations it touches are in one object. Then, by definition, the operation can be performed entirely at one n ode (reference tree mechanisms can handle cases where a single object to be updated has copies on several nodes).

The standard LISP operation **rplaca**(*X*;*Y*) replaces the **car** of conscell *X* with the value *Y*. In LISP, it is conventional for the **rplaca** function to return *X*, the modified

conscell. A simple change to this would be to have it return instead the old value of **car**(*X*). Then $Z \leftarrow$ **rplaca**(*X*, *Y*) would have the same meaning as the sequence

$$Z \leftarrow \mathbf{car}(X)$$

$$\mathbf{car}(X) \leftarrow Y$$

executed as one atomic operation. (A complementary **rplacd** operation can obviously be defined in an analogous manner.) This atomic **rplaca**, which is a single-object (in fact, *single-location*) operation, is a variant of the conventional test-and-set operation. It can clearly be used to implement, *e.g.*, semaphores, by busy-waiting, but in fact it makes possible the implementation of both fork/join and fair semaphores with no busy-waiting. Details are given in [1].

The LISP-based stack machine described above is currently being used for a myriaprocessor design to be tested via simulation and ultimately constructed out of VLSI integrated circuits. The success of projects such as the SCHEME chip [11] are encouraging indications that such a plan of attack is realistic. The existence of efficient synchronization operators based on single-object atomic operations (such as the **rplaca** and **rplacd** described above) is a reason to believe that our approach has not built-in logical limitations in dealing with concurrent processes.

Accomplishments during the past year include:

1) Specification of an initial version of the low-level LCODE instruction set.

2) Construction of a compiler from a LISP dialect into LCODE.

3) Construction of a simulator for executing LCODE programs and measuring their behavior in a hypothetical MuNet.

4) Specification of several candidate architectures for MuNet nodes.

5) Development of an approach, based on execution traces, for studying the "working set" behavior of LCODE programs, to help develop and evaluate object and task distribution heuristics and estimate requirements for node size and communication bandwidth.

Among the projects planned for the future are the following:

1) Detailed design of the low-level LCODE instruction set. A balance must be struck between the complexity of individual nodes, execution speed, and memory requirements.

2) Reasonably concrete specification of a MuNet node architecture. This will include design of hardware for efficiently handling communication between nodes, using reference trees or other protocols. Ideally, communication of objects and tasks between neighboring nodes should occur at speeds rivalling those of processor-memory buses on contemporary machines.

3) Development of strategies for the distribution of tasks and data. This endeavor should benefit from the "working set" studies mentioned above.

4) To reduce hardware complexity and increase the flexibility of a MuNet node design, it is preferable not to implement the distribution *strategy* in hardware, but supply only the minimal necessary "hooks" for distribution strategies implemented in *software* to gather needed statistics about system operation and effect the adjustments called for. The design of these "hooks" is another important challenge.

5) Development of realistic application programs for the MuNet, and collection (via simulation) of numerical performance figures for them.

To discover the basic capabilities and properties of myriaprocessor architectures, it is appropriate to tackle simple problems and simple projects first. Much can be learned about scalability and distribution strategies, for example, using just the simple LISP model of data and a small, basic instruction set. A production-quality machine would have additional data types and instructions to mitigate the overhead of manipulating objects as small as conscells, in steps as small as the simple stack-machine instructions. Nevertheless, the LISP-inspired object-orientation and the basic separation between hardware bookkeeping and software strategy functions would form the philosophical underpinnings of a production architecture as well.

5. VLSI DESIGN TOOLS

The main emphasis of this effort continued to be the development of LSI design and verification tools running in UNIX environment. New versions of several major components in the system were completed:

- Algorithmic improvements were made in the program that extracts electrical networks from mask information. The new algorithms use much less mass storage and offer an order-of-magnitude speed-up over the older versions.

- The electrical rules checker was enhanced to provide a more complete static analysis for MOS circuits.

- A cell-based graphics editor was brought up on the Nu personal computer system under V7 UNIX using a 512x512 8-bits-per-pixel raster-scan color display.

- A LISP-like front end for ESIM (a switch-level logic simulator) was completed which allows the designer to easily construct sophisticated simulation environments with which to test his design.

In cooperation with Digital Equipment Corporation, the suite of verification tools was used to successfully process a large commercial HMOS design. Experience gained in processing this and other large designs led to the improvements outlined above.

Work on a new simulation algorithm was begun. In the new simulator transistors are modeled as resistors whose resistance is determined by the voltage on the terminal nodes. Networks of transistors and electrical nodes form an R-C tree (R from the transistors, C from the interconnect) under this model; the network's behavior under different inputs is calculated by an event-driven simulator. The comparatively fast "pseudo circuit analysis" provided by the new simulator allows the designer to determine both the functional and timing characteristics of his network with more accuracy than switch-level simulation, using larger circuits than can be accommodated by circuit analysis programs.

Work has also started on integrating our tools into a more complete design system. A prototype text-based schematic entry system was constructed and work is underway on automatic generation of schematic diagrams from the data base. A network isomorphism program was developed that allows comparison of an entered schematic with that derived from the layout.

Chip design projects within the group have provided a test bed for these tools, but may be of independent interest. Among these is a new device developed by Johnson and Zippel to offer a unique memory management scheme for the Nu; the chip is a 32 word by 24 bit, fully associative, content addressable memory. It provides a mapping between 24-bit virtual addresses and 5-bit physical page ID's. One principle was uppermost in the electrical design of the chip: it should work correctly no matter how poorly controlled the wafer processing was. This led to several deviations from Mead/Conway design methods:

- The average "inverter ratio" was about 8, not the M/C standard of 4.0.

- There are no minimum-geometry transistors on the chip.

- TTL-level inputs were amplified to full internal signal swings by inverters with ratio 21.

- The output pad drivers were designed to work into a load of 200 pF.

- The memory cells are unclocked, fully static flip-flops.

- The wired-OR match lines are precharged to {Vdd - (2.0*Vt)}, not Vdd-Vt.

- A two-section substrate bias generator was included on the chip.

- Simulations indicate that the chip will work properly with enhancement threshold of 1.1 volt plus or minus 0.5 volts, and depletion threshold of -5.0 volts, plus or minus 2.0 volts.

- The simulated "access time" for an associative lookup (with nominal processing parameters) is 80 nanoseconds.

- It appears reasonable to build a CAM with 4 times as many bits on an MPC process; if we could get a well-controlled, well-characterized industry process like HMOS-I or HMOS-II, 8 times as many bits (naively) seem feasible.

References

1. Halstead, R. "Architecture of a Myriaprocessor," Proceedings COMPCON, San Francisco, CA, February 1981.

2. Webster's New World Dictionary of the American Language, New York, NY, 1970, The World Publishing Company.

3. Arvind, Gostelow, K. and Plouffe, W. "An Asynchronous Programming Language and Computing Machine", University of California at Irvine Report TR114a, Irvine, CA, 1978.

4. Ackerman, W. and Dennis, J. "VAL- A Value-Oriented Algorithmic Language", MIT/LCS TR-218, MIT Laboratory for Computer Science, Cambridge, MA, 1979.

5. Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages," Working Paper 92, MIT Artificial Intelligence Laboratory, Cambridge, MA, April 1976.

6. McCarthy, J. et al. LISP 1.5 Programmer's Manual, Cambridge, MA, MIT Press, 1962.

7. Greenblatt, R. "The LISP Machine," Working Paper 79, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1974.

8. Hansen, W. J. "Compact List Representation: Definition, Garbage Collection, and System Implementation," CACM 12, 9 (September 1969).

9. Halstead, R. "Reference Tree Networks: Virtual Machine and Implementation," MIT/LCS/TR-222, MIT Laboratory for Computer Science, Cambridge, MA, 1979.

10. Halstead, R. and Ward, S. "The MuNet: A Scalable Decentralized Architecture for Parallel Computation," Seventh Annual Symposium on Computer Architecture, La Baule, France, May 1980.

11. Steele, G. and G. Sussman "Design of a LISP-Based Microprocessor," CACM 23, 11 (November 1980).

Publications

1. Baker, C. and Terman, C. "Tools for Verifying Integrated Circuit Design," Lambda, 4th quarter, (1980).

2. Halstead, R. H. "Architecture of a Myriaprocessor," Proceedings COMPCON Spring 81, San Francisco, CA, February 1981.

Talks

1. Dertouzos, M. L. "The Next Twenty Years with Computers and People," Siemens Corporation Munich, Germany, November 1980.

2. Dertouzos, M. L. "The Information Marketplace," AFIPS Workshop on Technical and Policy Issues in Electronic Mail and Message Systems, Washington, DC, December 1980.

3. Dertouzos, M. L., "Computers and People: Future Partnership or Conflict?" Inaugural Symposia for MIT President Paul E. Gray, Cambridge, MA, September 1981.

4. Dertouzos, M. L. "The New Relationship Between the U.S. and Japan Electronics," Electro81, New York, April 1981.

5. Dertouzos, M. L. "Long Range Research at the MIT Laboratory for Computer Science," NEC Systems Laboratory, Inc., Lexington, MA, April 1981.

6. Dertouzos, M. L. "Overview and Future Trends," Research Highlights at the MIT Laboratory for Computer Science, MIT Industrial Liaison Symposium, Cambridge, MA, May 1981.

7. Dertouzos, M. L. "Secrecy and Cryptography Research," National Engineering Society and MIT Science, Technology and Society Program, Cambridge, MA, May 1981.

8. Dertouzos, M. L. "Expected Technological Developments in the Computer Field: The Information Marketplace," MIT Industrial Liaison Program European Course Personal Computers, Networks, and Office Automation, Zurich, Switzerland, June 1981.

9. Ward, S. "Introduction to Personal Computers," MIT Industrial Liaison

Program European Course, Personal Computers, Networks, and Office Automation, Zurich, Switzerland, June 1981.

10. Ward, S. "Future Architecture of Personal Computers," MIT Industrial Liaison Program European Course, Personal Computers, Networks, and Office Automation, Zurich, Switzerland, June 1981.

11. Ward, S. "Personal Computing Environments," MIT Industrial Liaison Program European Course, Personal Computers, Networks, and Office Automation, Zurich, Switzerland, June 1981.

12. Ward, S. "Personal Computers and Distributed Operating Systems," MIT Industrial Liaison Program Symposium, Cambridge, MA, May 1981.

13. Ward, S. "Personal Computer Systems," Siemens AG, Munich, Germany, November 1980.

14. Ward, S. "Multiprocessor Architectures," COMPCON 81, San Francisco, CA, February 1981.

Theses Completed

1. Arnold, J. M. "A Floating Point Array Processor for the Nu Computer," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

2. Berger, N. "A Software Controller for Characterizing the T-11 Microprocessor Chip," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

3. Eliot, C. "An Automatic Telephone Directory and Dialer," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

4. Freidah, J. "A Pascal to Fortran Translator," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

5. George, P. "Performance Analysis of Real-Time Systems," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

6. Hu, A. "A Field Management System for a Display Data Terminal," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

Theses in Progress

1. Goddeau, D. "Automatic Schematic Generation for a Language-based Computer Aided Circuit Design System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected Spring 1982.

2. Johnson, M. "Exploiting Interface Specifications in Hierarchical VLSI Design," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1982.

3. Mok, A. "Fundamental Design Problems of Multiprocessor Systems for Hard Real-Time Environments," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1982.

4. Powell, J. "Voice Entry for Interactive Control of a Small Computer," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected Spring 1982.

5. Sieber, J. "TRIX: An Operating System Supporting Network Communications," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1982.

6. Simmons, R. "Hidden Surface Removal Through Object Space Decomposition," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1982.

7. Teixeira, T. J. "Compiling Programs to Meet Performance Requirements," Ph.D dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected Spring 19[illegible]

8. Terman, C.J. "Simulation Tools for LSI Design," Ph.D dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected Spring 1982.

9. Troisi, J. "A Compiler and Debugger for C," S.M. thesis, MIT Department Electrical Engineering and Computer Science, Cambridge, MA, expected May 1982.

SYSTEMATIC PROGRAM DEVELOPMENT

Academic Staff

J. Guttag, Group Leader

Graduate Students

S. Atreya
S. Manayam
A. Wells
J. Wing
J. Zachary

Undergraduate Students

R. Forgaard
J. Goree

Support Staff

E. Pothier

Postdoctoral Fellow

P. Lescanne

Systematic Program Development

1. INTRODUCTION

Our research over the past year has been concentrated in four areas: an investigation of the uses of formal specifications, the development of a system to aid in the construction and use of formal specifications, rewrite rule theory, and programming languages. The first three of these are closely related, and we expect them to continue to dominate our work over the next few years. The last is not an area we expect to spend much time on in the immediate future.

2. FORMAL SPECIFICATION OF SOFTWARE

2.1. How to Use Specifications

Over the last few years specifications in general, and formal specifications in particular, have become "hot" topics of research. Most of the work in this area has been centered around one or another aspect of the presentation and evaluation of a particular specification language or class of specification languages. Over the last year we have tried to step back and take a broad view of the role of formal specifications in the program development process. This view is an outgrowth of problems encountered in trying to extend and apply our earlier work on specifications.

As our understanding of the theoretical and linguistic aspects of formal specifications improved, we began to try to use them in developing interesting software. We ran into serious problems doing this. Not because we encountered things that we found difficult to specify, but because we were unsure about what we wanted to do with the specifications we were writing. i.e., we were uncertain about how formal specifications should fit into the ongoing process of developing relatively large programs. This uncertainty about the exact uses to which our specifications would and should be geared, led to an inconsistent attitude about what was or was not a "good" specification. We feel that over the last year we have made considerable progress in sorting these things out. One of the points is that there are at least two distinct uses for specifications. They can be used to describe the behavior that a program is supposed to exhibit or they can describe properties of the program's structure. Randy Forgaard [1] explored this issue by constructing and using in different ways different specifications of the same program.

In order to formalize our analysis of specifications, Jeannette Wing developed a preliminary formal model of a large specification. A specification is a graph where each node is a specification of a data type and arcs denote a dependency relation. From this representation, we define many structural relations found among types specifications, subsetting relations found between two large specifications, and properties of specifications with respect to specificands [2][3] . Given a formal model of a specification, we plan to analyze the relations within and among specifications and to check for whether properties hold as a specification is developed and after it is completed.

2.2. A Specification Environment

Our experience in writing and using specifications has led us to the conclusion that if formal specifications are to become practical tools it will be necessary to have available a sophisticated system to help specifiers construct, read, and analyze specifications. The design and implementation of such a system has provided and we expect will continue to provide a focal point for much of our group's research.

A sophisticated data base or library of specifications and partial specifications will play an important role in this system. Sriram Atreya has just begun work on the design of that library. Some preliminary observations about the functionality of the library are contained in [4]. Besides serving as a simple file-system for specifications, the library will keep track of various dependency relations among these specifications and other auxiliary information that may prove useful to a user while developing specifications. An important component of the library will be various mechanisms, including a browser, designed to help a user find things contained in the library.

Joe Zachary has been working on the design and specification of a specification editor. The specification editor will provide an interface between the user and the specification library. Since knowledge of the specification language will be built into it, the editor will provide syntax-directed editing capabilities. Further, the editor will enable the user to update the library with new type definitions and to browse through the library. In order to specify the editor it is necessary to first specify the specification language. In doing this, various extensions to the specification language are proposed. These are described in [5].

The initial implementation of the specification environment will run partly on a DEC 20 and partly on a Xerox Alto. Most of the computation will be done on the 20, with the Alto serving primarily as a sophisticated terminal. Allen Wells has been working on connecting an Alto to a 20 using an Alto EIA board. He has also written a general window package for the Alto upon which we expect to build part of the user interface to our system.

2.3. Example Specifications

We feel strongly that a significant problem in work on specifications has been a lack of interesting medium and large size examples. We have, therefore, continued to work on the development of examples.

Jeannette Wing completed the specification of the simple banking system started last year, and Sriram Atreya successfully implemented it working directly from the specification. The specification, implementation, and various observations about their experiences with this example are described in [4].

Randy Forgaard has completed the specification and implementation of a window package for the alto display. Once he mastered our specification technique, the specification proceeded smoothly and quickly. In going from the initial specification we learned a number of things about the relationship of behavioral specifications to structural specifications and about the relationship of structural specifications to implementations. Some time was also spent comparing formal specifications to informal ones. This work is described in [1].

3. REWRITE RULE THEORY

Term rewriting systems, also called rewrite rule systems, is a model of computation that has the interesting and useful property of being directly applicable to obtaining decision procedures for equational theories. Equational theories, in turn, supply the formal basis of our approach to specification.

In the past year, John Guttag has worked closely with Dave Musser and Deepak Kapur of General Electric Corporate Research and Development on developing methods for proving the finite termination of a set of rewrite rules. The conventional approach to this problem involves constructing a mapping from the terms to be rewritten onto a well-founded partially ordered set. Guttag, Kapur and Musser take a radically different approach that, while not as general as other approaches, is more algorithmic. This method will usually not yield a complete proof of finite termination by itself, but can be used to simplify the application of other methods. The method is described in [6].

In three papers, Pierre Lescanne studied various aspects of an algorithm that proves termination of rewrite systems. The first addresses the well-foundedness property of the "Decomposition Ordering" [7]. It also shows that when the ordering on basic operator symbols is total, the Decomposition Ordering and the Recursive Path Ordering proposed by Dershowitz [8] are equivalent. The second paper compares the complexities of an implementation derived from the Decomposition Ordering and one derived from Dershowitz's Recursive Path Ordering [9]. The

comparison is made for the case of monadic terms on a totally ordered set of symbols. Average case analysis shows that Decomposition Ordering leads to a better implementation in general. The third paper proposes a data structure to implement Decomposition Ordering in the general case [10].

John Goree has begun work on a prototype of a Rewrite Rule Laboratory -- a software "environment" which provides many of the basic data structures and algorithms of rewrite rule theory. Using an existing data abstraction for terms (based on tree manipulation) and a few existing procedures for operations such as substitution and extracting subterms, he has implemented additional primitive operations for the Laboratory (e.g., unification).

As a study in the effectiveness of the Rewrite Rule Lab as a foundation for development, John Goree has implemented the Knuth-Bendix completion procedure using the primitive operations of the prototype Lab. He has investigated a modification to the original Knuth-Bendix procedure; this technique essentially attempts to "complete" a set of rewrite rules by adding all "critical pairs" to the rule set at once, rather than adding them one at a time as does the original method as proposed by Knuth and Bendix. Further analysis of this modified procedure is needed to answer questions of efficiency and convergence to a complete set. This work is described in [11].

4. SYNTHESIS OF IMPLEMENTATIONS OF DATA ABSTRACTIONS

M.K. Srivas has been studying the problem of automatically synthesizing implementations for abstract data types starting from the algebraic specifications of the data types. This research will be presented in his Ph.D. dissertation [12] which he expects to complete this summer.

In this method the user specifies the data types involved, the implemented (or abstract) type and the implementing (or concrete) types, algebraically. He also gives an algebraic description of a function showing how the values of the concrete types are used to represent the values of the abstract type. (This function is similar to Hoare's abstraction function [13].) Using this information, the system generates implementations for each of the operations of the abstract type in terms of the operations of the concrete types.

The above problem is, in general, unsolvable since it involves solving the word problem which is known to be unsolvable [14]. So, we impose restrictions on the form of the input specifications, and also assume some knowledge about the structure of the output the user expects. The two major goals of the research are (i) to develop and study the principle behind a synthesis method that works in a

reasonable number of cases, and (ii) to formally characterize the conditions under which the method works.

The approach of the synthesis method is roughly the reverse of program verification. In program verification, given a specification an implementation and a proof technique, we attempt to prove a theorem that establishes the consistency between the theorem and the implementation. In program synthesis, given a specification, and a proof technique, we try to generate a set of theorems (that were proved using the proof technique) from which a suitable implementation could be extracted. The proof technique employed is based on rewrite rule theory. We investigate equational proof technique as well as a class of inductive proof techniques in synthesizing implementations.

5. PROGRAMMING LANGUAGES

5.1. Typed Functional Programming

John Guttag has worked with John Williams of IBM Research on the introduction of data abstraction and strong typing into functional programming. As a result of this work, John Guttag is writing a report which describes in abstract terms why it would be useful to incorporate data abstraction and strong typing into functional programming languages. A paper describing how to incorporate these into Backus' functional programming language is currently under preparation.

5.2. Type Inference

Type inference is used to try to obtain some of the advantages of strong typing without having to specify the types of variables. In [15], Allen Wells describes work done with Jim Morris of Xerox PARC on the design and implementation of a type inference system for Mesa, a strongly typed language with a very rich domain of types. The goal was to relax some of the typing restrictions currently in the language without sacrificing any of the advantages of strong typing. The algorithm used works well for the majority of the types in Mesa, but sometimes fails when dealing with types that are not symmetric with respect to assignment (the assignment of an integer to a real is permitted in Mesa, but the assignment of a real to an integer is not, for example). Two solutions to this problem were explored: the first involves an extremely inefficient type inference algorithm and the second, modifications to the language definition.

6. INTERACTIONS OUTSIDE OF LCS

The Systematic Program Development Group has made a point of maintaining close contact with researchers outside of MIT. Over the past year, John Guttag and Srivas Mandayam have worked with Dave Musser and Deepak Kapur of General Electric Corporate Research and Development, John Guttag and Jeannette Wing have worked with Jim Horning of Xerox PARC, Allen Wells has worked with Jim Morris of Xerox PARC, and John Guttag has worked with John Williams of IBM Research at San Jose.

References

1. Forgaard, R. "Applying a Formal Program Specification Methodology," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1981.

2. Wing, J.M. "Formal Model of a Requirements Specification Graph: Basic Definitions," Systematic Program Development Group Internal Memo 2, MIT Laboratory for Computer Science, Cambridge, MA, January 1981.

3. Wing, J.M. "More Definitions," Systematic Program Development Internal Memo 3, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

4. Atreya, S.K. and Wing, J.M. "Design and Implementation of a System Using Formal Specifications," Systematic Program Development Group Working Paper 4, MIT Laboratory for Computer Science, Cambridge, MA, March 1981.

5. Zachary, J.L., Wing, J.M. and Atreya, S.K. "Editor/Library Interface Preliminary Design Decisions," Systematic Program Development Group Internal Memo 5, MIT Laboratory for Computer Science, Cambridge, MA, May 1981.

6. Guttag, J., Kapur, D. and Musser, D. "On Proving Uniform Termination and Restricted Termination of Rewriting Systems," MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

7. Lescanne, P. "Decomposition Ordering as a Tool to Prove Termination of Writing Systems," Systematic Program Development Group Working Paper 5, MIT Laboratory for Computer Science, Cambridge, MA, October 1980.

8. Dershowitz, N. "Ordering for Term Rewriting System," Proceedings of 20th Symposium on Foundations of Computer Science, 1979, 123-131.

9. Lescanne, P. "Two Implementations of Recursive Path Ordering on Monadic Terms," Systematic Program Development Group Working Paper 6, MIT Laboratory for Computer Science, Cambridge, MA, December 1980.

10. Lescanne, P., "How Does Recursive Path Ordering Work?" Systematic

Program Development Working Paper 7, MIT Laboratory for Computer Science, Cambridge, MA, March 1981.

11. Goree, J. "Using Abstractions to Implement the Knuth-Bendix Completion Procedure," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1981.

12. Srivas, M.K., "Synthesis Implementations for Abstract Data Types", Ph.D. Dissertation (Forthcoming), MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

13. Hoare, C.A.R. "Proof of Correctness of Data Representations," Acta Informatica, 1, 4, (1972), 271-281.

14. Knuth, D.E. and Bendix P.B. "Simple Word Problems in Universal Algebras," in Computational Problems in Abstract Algebra, Leech J. (Ed.), Pergamon Press, 1970, 263-297.

15. Wells, A. "Type Inference in Strongly Typed Languages," M.S. thesis MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1981

Theses in Progress

1. Forgaard, R. "Applying a Formal Program Specification Methodology," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1981.

2. Goree, J. "Using Abstractions to Implement the Knuth-Bendix Completion Procedure," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1981.

3. Srivas, M.K., "Synthesis Implementations for Abstract Data Types", Ph.D. Dissertation (Forthcoming), MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1981.

4. Wells, A. "Type Inference in Strongly Typed Languages," M.S. thesis Cambridge, MA, expected May 1981. MIT Department of Electrical Engineering and Computer Science.

PUBLICATIONS

Technical Memoranda

TM-10[10] Jackson, James N.
Interactive Design Coordination for the Building Industry, June 1970, AD 708-400

TM-11 Ward, Philip W.
Description and Flow Chart of the PDP-7/9 Communications Package, July 1970, AD 711-379

TM-12 Graham, Robert M.
File Management and Related Topics June 12, 1970, September 1970, AD 712-068

TM-13 Graham, Robert M.
Use of High Level Languages for Systems Programming, September 1970, AD 711-965

TM-14 Vogt, Carla M.
Suspension of Processes in a Multi-processing Computer System, September 1970, AD 713-989

TM-15 Zilles, Stephen N.
An Expansion of the Data Structuring Capabilities of PAL, October 1970, AD 720-761

TM-16 Bruere-Dawson, Gerard
Pseudo-Random Sequences, October 1970, AD 713-852

TM-17 Goodman, Leonard I.
Complexity Measures for Programming Languages, September 1971, AD 729-011

TM-18 Reprinted as TR-85

TM-19 Fenichel, Robert R.
A New List-Tracing Algorithm, October 1970, AD 714-522

---

[10] TMs 1-9 were never issued.

TM-20 Jones, Thomas L.
A Computer Model of Simple Forms of Learning, January 1971, AD 720-337

TM-21 Goldstein, Robert C.
The Substantive Use of Computers For Intellectual Activities, April 1971, AD 721-618

TM-22 Wells, Douglas M.
Transmission Of Information Between A Man-Machine Decision System And Its Environment, April 1971, AD 722-837

TM-23 Strnad, Alois J.
The Relational Approach to the Management of Data Bases, April 1971, AD 721-619

TM-24 Goldstein, Robert C. and Alois J. Strnad
The MacAIMS Data Management System, April 1971, AD 721-620

TM-25 Goldstein, Robert C.
Helping People Think, April 1971, AD 721-998

TM-26 Iazeolla, Giuseppe G.
Modeling and Decomposition of Information Systems for Performance Evaluation, June 1971, AD 733-965

TM-27 Bagchi, Amitava
Economy of Descriptions and Minimal Indices, January 1972, AD 736-960

TM-28 Wong, Richard
Construction Heuristics for Geometry and a Vector Algebra Representation of Geometry, June 1972, AD 743-487

TM-29 Hossley, Robert and Charles Rackoff
The Emptiness Problem for Automata on Infinite Trees, Spring 1972, AD 747-250

TM-30 McCray, William A.
SIM360: A S/360 Simulator, October 1972, AD 749-365

TM-31 Bonneau, Richard J.
A Class of Finite Computation Structures Supporting the Fast Fourier Transform, March 1973, AD 757-787

TM-32 Moll, Robert
An Operator Embedding Theorem for ComplexityClasses of Recursive Functions, May 1973, AD 759-999

TM-33 Ferrante, Jeanne and Charles Rackoff
A Decision Procedure for the First Order Theory of Real Addition with Order, May 1973, AD 760-000

TM-34 Bonneau, Richard J.
Polynomial Exponentiation: The Fast Fourier Transform Revisited, June 1973, PB 221-742

TM-35 Bonneau, Richard J.
An Interactive Implementation of the Todd-Coxeter Algorithm, December 1973, AD 770-565

TM-36 Geiger, Steven P.
A User's Guide to the Macro Control Language, December 1973, AD 771-435

TM-37 Schonhage, A.
Real-Time Simulation of Multidimensional Turing Machines by Storage Modification Machines, December 1973, PB 226-103/AS

TM-38 Meyer, Albert R.
Weak Monadic Second Order Theory of Succesor is not Elementary-Recursive, December 1973, PB 226-514/AS

TM-39 Meyer, Albert R.
Discrete Computation: Theory and Open Problems, January 1974, PB 226-836/AS

TM-40 Paterson, Michael S., Michael J. Fischer and Albert R. Meyer
An Improved Overlap Argument for On-Line Multiplication, January 1974, AD 773-137

TM-41 Fischer, Michael J. and Michael S. Paterson
String-Matching and Other Products, January 1974, AD 773-138

TM-42 Rackoff, Charles
On the Complexity of the Theories of Weak Direct Products, January 1974, PB 228-459/AS

TM-43 Fischer, Michael J. and Michael O. Rabin
Super-Exponential Complexity of Presburger Arithmetic, February 1974, AD 775-004

TM-44 Pless, Vera
Symmetry Codes and their Invariant Subcodes, May 1974, AD 780-243

TM-45 Fischer, Michael J. and Larry J. Stockmeyer
Fast On-Line Integer Multiplication, May 1974, AD 779-889

TM-46 Kedem, Zvi M.
Combining Dimensionality and Rate of Growth Arguments for Establishing Lower Bounds on the Number of Multiplications, June 1974, PB 232-969/AS

TM-47 Pless, Vera
Mathematical Foundations of Flip-Flops, June 1974, AD 780-901

TM-48 Kedem, Zvi M.
The Reduction Method for Establishing Lower Bounds on the Number of Additions, June 1974, PB 233-538/AS

TM-49 Pless, Vera
Complete Classification of (24,12) and (22,11) Self-Dual Codes, June 1974, AD 781-335

TM-50 Benedict, G. Gordon
An Enciphering Module for Multics, S.B. Thesis, EE Dept., July 1974, AD 782-658

TM-51 Aiello, Jack M.
An Investigation of Current Language Support for the Data Requirements of Structured Programming, S.M. & E.E. Thesis, EE Dept., September 1974, PB 236-815/AS

TM-52 Lind, John C.
Computing in Logarithmic Space, September 1974, PB 236-167/AS

TM-53 Bengelloun, Safwan A.
MDC-Programmer: A Muddle-to Datalanguage Translator for Information Retrieval, S.B. Thesis, EE Dept., October 1974, AD 786-754

TM-54 Meyer, Albert R.
The Inherent Computation Complexity of Theories of Ordered Sets: A Brief Survey, October 1974, PB 237-200/AS

TM-55 Hsieh, Wen N., Larry H. Harper and John E. Savage
A Class of Boolean Functions with Linear Combinatorial Complexity, October 1974, PB 237-206/AS

TM-56 Gorry, G. Anthony
Research on Expert Systems, December 1974

TM-57 Levin, Michael
On Bateson's Logical Levels of Learning, February 1975

TM-58 Qualitz, Joseph E.
Decidability of Equivalence for a Class of Data Flow Schemas, March 1975, PB 237-033/AS

TM-59 Hack, Michel
Decision Problems for Petri Nets and Vector Addition Systems, March 1975 PB 231-916/AS

TM-60 Weiss, Randell B.
CAMAC: Group Manipulation System, March 1975, PB 240-495/AS

TM-61 Dennis, Jack B.
First Version of a Data Flow Procedure Language, May 1975

TM-62 Patil, Suhas S.
An Asynchronous Logic Array, May 1975

TM-63 Pless, Vera
Encryption Schemes for Computer Confidentiality, May 1975, AD A010-217

TM-64 Weiss, Randell B.
Finding Isomorph Classes for Combinatorial Structures, S.M. Thesis, EE Dept., June 1975

TM-65 Fischer, Michael J.
The Complexity Negation-Limited Networks - A Brief Survey, June 1975

TM-66 Leung, Clement
Formal Properties of Well-Formed Data Flow Schemas, S.B., S.M. & E.E. Thesis, EE Dept., June 1975

TM-67 Cardoza, Edward E.
Computational Complexity of the Word Problem for Commutative Semigroups, S.M. Thesis, EE & CS Dept., October 1975

TM-68 Weng, Kung-Song
Stream-Oriented Computation in Recursive Data Flow Schemas, S.M. Thesis, EE & CS Dept., October 1975

TM-69 Bayer, Paul J.
Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees, S.M. Thesis, EE & CS Dept., November 1975

TM-70 Ruth, Gregory R.
Automatic Design of Data Processing Systems, February 1976, AD A023-451

TM-71 Rivest, Ronald
On the Worst-Case of Behavior of String-Searching Algorithms, April 1976

TM-72 Ruth, Gregory R.
Protosystem I: An Automatic Programming System Prototype, July 1976, AD A026-912

TM-73 Rivest, Ronald
Optimal Arrangement of Keys in a Hash Table, July 1976

TM-74 Malvania, Nikhil
The Design of a Modular Laboratory for Control Robotics, S.M. Thesis, EE & CS Dept., September 1976, AD A030-418

TM-75 Yao, Andrew C. and Ronald I. Rivest
K + 1 Heads are Better than K, September 1976, AD A030-008

TM-76 Bloniarz, Peter A., Michael J. Fischer and Albert R. Meyer
A Note on the Average Time to Compute Transitive Closures, September 1976

TM-77 Mok, Aloysius K.
Task Scheduling in the Control Robotics Environment, S.M. Thesis, EE & CS Dept., September 1976, AD A030-402

TM-78 Benjamin, Arthur J.
Improving Information Storage Reliability Using a Data Network, S.M. Thesis, EE & CS Dept., October 1976, AD A033-394

TM-79 Brown, Gretchen P.
A System to Process Dialogue: A Progress Report, October 1976, AD A033-276

TM-80 Even, Shimon
The Max Flow Algorithm of Dinic and Karzanov: An Exposition, December 1976

TM-81 Gifford, David K.
Hardware Estimation of a Process' Primary Memory Requirements, S.B. Thesis, EE & CS Dept., January 1977

TM-82 Rivest, Ronald L., Adi Shamir and Len Adleman
A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, April 1977, AD A039-036

TM-83 Baratz, Alan E.
Construction and Analysis of Network Flow Problem which Forces Karzanov Algorithm to $O(n^3)$ Running Time, April 1977

TM-84 Rivest, Ronald L. and Vaughan R. Pratt
The Mutual Exclusion Problem for Unreliable Processes, April 1977

TM-85 Shamir, Adi
Finding Minimum Cutsets in Reducible Graphs, June 1977, AD A040-698

TM-86 Szolovits, Peter, Lowell B. Hawkinson and William A. Martin
An Overview of OWL, A Language for Knowledge Representation, June 1977, AD A041-372

TM-87 Clark, David., editor
Ancillary Reports: Kernel Design Project, June 1977

TM-88 Lloyd, Errol L.
On Triangulations of a Set of Points in the Plane, S.M. Thesis, EE & CS Dept., July 1977

TM-89 Rodriguez, Humberto Jr.
Measuring User Characteristics on the Multics System, S.B. Thesis, EE & CS Dept., August 1977

TM-90 d'Oliveira, Cecilia R.
An Analysis of Computer Decentralization, S.B. Thesis, EE & CS Dept., October 1977, AD A045-526

TM-91 Shamir, Adi
Factoring Numbers in O(log n) Arithmetic Steps, November 1977, AD A047-709

TM-92 Misunas, David P.
Report on the Workshop on Data Flow Computer and Program Organization, November 1977

TM-93 Amikura, Katsuhiko
A Logic Design for the Cell Block of a Data-Flow Processor, S.M. Thesis, EE & CS Dept., December 1977

TM-94 Berez, Joel M.
A Dynamic Debugging System for MDL, S.B. Thesis, EE & CS Dept., January 1978, AD A050-191

TM-95 Harel, David
Characterizing Second Order Logic with First Order Quantifiers, February 1978

TM-96 Harel, David, Amir Pnueli and Jonathan Stavi
A Complete Axiomatic System for Proving Deductions about Recursive Programs, February 1978

TM-97 Harel, David, Albert R. Meyer and Vaughan R. Pratt
Computability and Completeness in Logics of Programs, February 1978

TM-98 Harel, David and Vaughan R. Pratt
Nondeterminism in Logics of Programs, February 1978

TM-99 LaPaugh, Andrea S.
The Subgraph Homeomorphism Problem, S.M. Thesis, EE & CS Dept., February 1978

TM-100 Misunas, David P.
A Computer Architecture for Data-Flow Computation, S.M. Thesis, EE & CS Dept., March 1978, AD A052-538

TM-101 Martin, William A.
Descriptions and the Specialization of Concepts, March 1978, AD A052-773

TM-102 Abelson, Harold
Lower Bounds on Information Transfer in Distributed Computations, April 1978

TM-103 Harel, David
Arithmetical Completeness in Logics of Programs, April 1978

TM-104 Jaffe, Jeffrey
The Use of Queues in the Parallel Data Flow Evaluation of "If-Then-While" Programs, May 1978

TM-105 Masek, William J. and Michael S. Paterson
A Faster Algorithm Computing String Edit Distances, May 1978

TM-106 Parikh, Rohit
A Completeness Result for a Propositional Dynamic Logic, July 1978

TM-107 Shamir, Adi
A Fast Signature Scheme, July 1978, AD A057-152

TM-108 Baratz, Alan E.
An Analysis of the Solovay and Strassen Test for Primality, July 1978

TM-109 Parikh, Rohit
Effectiveness, July 1978

TM-110 Jaffe, Jeffrey M.
An Analysis of Preemptive Multiprocessor Job Scheduling, September 1978

TM-111 Jaffe, Jeffrey M.
Bounds on the Scheduling of Typed Task Systems, September 1978

TM-112 Parikh, Rohit
A Decidability Result for a Second Order Process Logic, September 1978

TM-113 Pratt, Vaughan R.
A Near-optimal Method for Reasoning about Action, September 1978

TM-114 Dennis, Jack B., Samuel H. Fuller, William B. Ackerman, Richard J. Swan and Kung-Song Weng
Research Directions in Computer Architecture, September 1978, AD A061-222

TM-115 Bryant, Randal E. and Jack B. Dennis
Concurrent Programming, October 1978, AD A061-180

TM-116 Pratt, Vaughan R.
Applications of Modal Logic to Programming, December 1978

TM-117 Pratt, Vaughan R.
Six Lectures on Dynamic Logic, December 1978

TM-118 Borkin, Sheldon A.
Data Model Equivalence, December 1978, AD A062-753

TM-119 Shamir, Adi and Richard E. Zippel
On the Security of the Merkle-Hellman Cryptographic Scheme, December 1978, AD A063-104

TM-120 Brock, Jarvis D.
Operational Semantics of a Data Flow Language, S.M. Thesis, EE & CS Dept., December 1978, AD A062-997

TM-121 Jaffe, Jeffrey
The Equivalence of R. E. Programs and Data Flow Schemes, January 1979

TM-122 Jaffe, Jeffrey
Efficient Scheduling of Tasks Without Full Use of Processor Resources, January 1979

TM-123 Perry, Harold M.
An Improved Proof of the Rabin-Hartmanis-Stearns Conjecture, S.M. & E.E. Thesis, EE & CS Dept., January 1979

TM-124 Toffoli, Tommaso
Bicontinuous Extensions of Invertible Combinatorial Functions, January 1979, AD A063-886

TM-125 Shamir, Adi, Ronald L. Rivest and Leonard M. Adleman
Mental Poker, February 1979, AD A066-331

TM-126 Meyer, Albert R. and Michael S. Paterson
With What Frequency Are Apparently Intractable Problems Difficult?, February 1979

TM-127 Strazdas, Richard J.
A Network Traffic Generator for Decnet, S.B. & S.M. Thesis, EE & CS Dept., March 1979

TM-128 Loui, Michael C.
Minimum Register Allocation is Complete in Polynomial Space, March 1979

TM-129 Shamir, Adi
On the Cryptocomplexity of Knapsack Systems, April 1979, AD A067-972

TM-130 Greif, Irene and Albert R. Meyer
Specifying the Semantics of While-Programs: A Tutorial and Critique of a Paper by Hoare and Lauer, April 1979, AD A068-967

TM-131 Adleman, Leonard M.
Time, Space and Randomness, April 1979

TM-132 Patil, Ramesh S.
Design of a Program for Expert Diagnosis of Acid Base and Electrolyte Disturbances, May 1979

TM-133 Loui, Michael C.
The Space Complexity of Two Pebble Games on Trees, May 1979

TM-134 Shamir, Adi
How to Share a Secret, May 1979, AD A069-397

TM-135 Wyleczuk, Rosanne H.
Timestamps and Capability-Based Protection in a Distributed Computer Facility, S.B. & S.M. Thesis, EE & CS Dept., June 1979

TM-136 Misunas, David P.
Report on the Second Workshop on Data Flow Computer and Program Organization, June 1979

TM-137 Davis, Ernest and Jeffrey M. Jaffe
Algorithms for Scheduling Tasks on Unrelated Processors, June 1979

TM-138 Pratt, Vaughan R.
Dynamic Algebras: Examples, Constructions, Applications, July 1979

TM-139 Martin, William A.
Roles, Co-Descriptors, and the Formal Representation of Quantified English Expressions (Revised May 1980), September 1979, AD A074-625

TM-140 Szolovits, Peter
Artificial Intelligence and Clinical Problem Solving, September 1979

TM-141 Hammer, Michael and Dennis McLeod
On Database Management System Architecture, October 1979, AD A076-417

TM-142 Lipski, Witold, Jr.
On Data Bases with Incomplete Information, October 1979

TM-143 Leth, James W.
An Intermediate Form for Data Flow Programs, S.M. Thesis, EE & CS Dept., November 1979

TM-144 Takagi, Akihiro
Concurrent and Reliable Updates of Distributed Databases, November 1979

TM-145 Loui, Michael C.
A Space Bound for One-Tape Multidimensional Turing Machines, November 1979

TM-146 Aoki, Donald J.
A Machine Language Instruction Set for a Data Flow Processor. S.M. Thesis, EE & CS Dept., December 1979

TM-147 Schroeppel, Richard and Adi Shamir
A $T = O(2^{n/2})$, $S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems, January 1980, AD A080-385

TM-148 Adleman, Leonard M. and Michael C. Loui
Space-Bounded Simulation of Multitape Turing Machines, January 1980

TM-149 Pallottino, Stefano and Tommaso Toffoli
An Efficient Algorithm for Determining the Length of the Longest Dead Path in an "Lifo" Branch-and-Bound Exploration Schema, January 1980, AD A079-912

TM-150 Meyer, Albert R.
Ten Thousand and One Logics of Programming, February 1980

TM-151 Toffoli, Tommaso
Reversible Computing, February 1980, AD A082-021

TM-152 Papadimitriou, Christos H.
On the Complexity of Integer Programming, February 1980

TM-153 Papadimitriou, Christos H.
Worst-Case and Probabilistic Analysis of a Geometric Location Problem, February 1980

TM-154 Karp, Richard M. and Christos H. Papadimitriou
On Linear Characterizations of Combinatorial Optimization Problems, February 1980

TM-155 Atai, Alon, Richard J. Lipton, Christos H. Papadimitriou and M. Rodeh
Covering Graphs by Simple Circuits, February 1980

TM-156 Meyer, Albert R. and Rohit Parikh
Definability in Dynamic Logic, February 1980

TM-157 Meyer, Albert R. and Karl Winklmann
On the Expressive Power of Dynamic Logic, February 1980

TM-158 Stark, Eugene W.
Semaphore Primitives and Starvation-Free Mutual Exclusion, S.M. Thesis, EE & CS Dept., March 1980

TM-159 Pratt, Vaughan R.
Dynamic Algebras and the Nature of Induction, March 1980

TM-160 Kanellakis, Paris C.
On the Computational Complexity of Cardinality Constraints in Relational Databases, March 1980

TM-161 Lloyd, Errol L.
Critical Path Scheduling of Task Systems with Resource and Processor Constraints, March 1980

TM-162 Marcum, Alan M.
A Manager for Named, Permanent Objects, S.B. & S.M. Thesis, EE & CS Dept., April 1980, AD A083-491

TM-163 Meyer, Albert R. and Joseph Y. Halpern
Axiomatic Definitions of Programming Languages: A Theoretical Assessment, April 1980

TM-164 Shamir, Adi
The Cryptographic Security of Compact Knapsacks (Preliminary Report), April 1980, AD A084-456

TM-165 Finseth, Craig A.
Theory and Practice of Text Editors or A Cookbook for an Emacs, S.B. Thesis, EE & CS Dept., May 1980

TM-166 Bryant, Randal E.
Report on the Workshop on Self-Timed Systems, May 1980

TM-167 Pavelle, Richard and Michael Wester
Computer Programs for Research in Gravitation and Differential Geometry, June 1980

TM-168 Greif, Irene
Programs for Distributed Computing: The Calendar Application, July 1980, AD A087-357

TM-169 Burke, Glenn and David Moon
LOOP Iteration Macro, (revised January 1981) July 1980, AD A087-372

TM-170 Ehrenfeucht, Andrzej, Rohit Parikh and Gregorz Rozenberg
Pumping Lemmas for Regular Sets, August 1980

TM-171 Meyer, Albert R.
What is a Model of the Lambda Calculus?, August 1980

TM-172 Paseman, William G.
Some New Methods of Music Synthesis, S.M. Thesis, EE & CS Dept., August 1980, AD A090-130

TM-173 Hawkinson, Lowell B.
XLMS: A Linguistic Memory System, September 1980, AD A090-033

TM-174 Arvind, Vinod Kathail and Keshav Pingali
A Dataflow Architecture with Tagged Tokens, September 1980

TM-175 Meyer, Albert R., Daniel Weise and Michael C. Loui
On Time Versus Space III, September 1980

TM-176 Seaquist, Carl R.
A Semantics of Synchronization, S.M. Thesis, EE & CS Dept., September 1980, AD A091-015

TM-177 Sinha, Mukul K.
TIMEPAD - A Performance Improving Synchronisation Mechanism for Distributed Systems, September 1980

TM-178 Arvind and Robert E. Thomas
I-Structures: An Efficient Data Type for Functional Languages, September 1980

TM-179 Halpern Joseph Y. and Albert R. Meyer
Axiomatic Definitions of Programming Languages, II, October 1980

TM-180 Papadimitriou, Christos H.
A Theorem in Database Concurrency Control, October 1980

TM-181 Lipski, Witold Jr. and Christos H. Papadimitriou
A Fast Algorithm for Testing for Safety and Detecting Deadlocks in Locked Transaction Systems, October 1980

TM-182 Hai, Alon, Christos H. Papadimitriou and Jayme Luiz Szwarefiter
Hamilton Paths in Grid Graphs, October 1980

TM-183 Meyer, Albert R.
A Note on the Length of Craig's Interpolants, October 1980

TM-184 Lieberman, Henry and Carl Hewitt
A Real Time Garbage Collector that can Recover Temporary Storage Quickly, October 1980

TM-185 Kung, Hsing-Tsung and Christos H. Papadimitriou
An Optimality Theory of Concurrency Control for Databases, November 1980, AD A092-625

TM-186 Szolovits, Peter and William A. Martin
BRAND X Manual, November 1980, AD A093-041

TM-187 Fischer, Michael J., Albert R. Meyer and Michael S. Paterson
CapOmega()(n log n) Lower Bounds on Length of Boolean Formulas, November 1980

TM-188 Mayr, Ernst
An Effective Representation of the Reachability Set of Persistent Petri Nets, January 1981

TM-189 Mayr, Ernst
Persistence of Vector Replacement Systems is Decidable, January 1981

TM-190 Ben-Ari, Mordechai, Joseph Y. Halpern and Amir Pnueli
Deterministic Propositional Dynamic Logic: Finite Models, Complexity, and Completeness, January 1981.

TM-191 Parikh, Rohit
Propositional Dynamic Logics of Programs: A Survey, January 198l.

TM-192 Meyer, Albert R., Robert S. Streett and Grazina Mirkowska
The Deducibility Problem in Propositional Dynamic Logic, February 1981

TM-193 Yannakakis, Mihalis and Christos H. Papadimitriou
Algebraic Dependencies, February 1981

TM-194 Barendregt, Henk and Giuseppe Longo
Recursion Theoretic Operators and Morphisms on Numbered Sets, February 1981

TM-195 Barber, Gerald R.
Record of the Workshop on Research in Office Semantics, February 1981

TM-196 Bhatt, Sandeep N.
On Concentration and Connection Networks, S.M. Thesis, EE & CS Dept., March 1981

Technical Reports

TR-1[11] Bobrow, Daniel G.
Natural Language Input for a Computer Problem Solving System, Ph.D. Dissertation, Math. Dept., September 1964, AD 604-730

TR-2 Raphael, Bertram
SIR: A Computer Program for Semantic Information Retrieval, Ph.D. Dissertation, Math. Dept., June 1964, AD 608-499

TR-3 Corbato, Fernando J.
System Requirements for Multiple-Access, Time-Shared Computers, May 1964, AD 608-501

TR-4 Ross, Douglas T. and Clarence G. Feldman
Verbal and Graphical Language for the AED System: A Progress Report, May 1964, AD 604-678

TR-6 Biggs, John M. and Robert D. Logcher
STRESS: A Problem-Oriented Language for Structural Engineering, May 1964, AD 604-679

TR-7 Weizenbaum, Joseph
OPL-1: An Open Ended Programming System within CTSS, April 1964, AD 604-680

TR-8 Greenberger, Martin
The OPS-1 Manual, May 1964, AD 604-681

TR-11 Dennis, Jack B.
Program Structure in a Multi-Access Computer, May 1964, AD 608-500

TR-12 Fano, Robert M.
The MAC System: A Progress Report, October 1964, AD 609-296

TR-13 Greenberger, Martin
A New Methodology for Computer Simulation, October 1964, AD 609-288

---

[11] Trs 5, 9, 10, 15 were never issued

TR-14 Roos, Daniel
Use of CTSS in a Teaching Environment, November 1964, AD 661-807

TR-16 Saltzer, Jerome H.
CTSS Technical Notes, March 1965, AD 612-702

TR-17 Samuel, Arthur L.
Time-Sharing on a Multiconsole Computer, March 1965, AD 462-158

TR-18 Scherr, Allan Lee
An Analysis of Time-Shared Computer Systems, Ph.D. Dissertation, EE Dept., June 1965, AD 470-715

TR-19 Russo, Francis John
A Heuristic Approach to Alternate Routing in a Job Shop, S.B. & S.M. Thesis, Sloan School, June 1965, AD 474-018

TR-20 Wantman, Mayer Elihu
CALCULAID: An On-Line System for Algebraic Computation and Analysis, S.M. Thesis, Sloan School, September 1965, AD 474-019

TR-21 Denning, Peter James
Queueing Models for File Memory Operation, S.M. Thesis, EE Dept., October 1965, AD 624-943

TR-22 Greenberger, Martin
The Priority Problem, November 1965, AD 625-728

TR-23 Dennis, Jack B. and Earl C. Van Horn
Programming Semantics for Multi-programmed Computations, December 1965, AD 627-537

TR-24 Kaplow, Roy, Stephen Strong and John Brackett
MAP: A System for On-Line Mathematical Analysis, January 1966, AD 476-443

TR-25 Stratton, William David
Investigation of an Analog Technique to Decrease Pen-Tracking Time in Computer Displays, S.M. Thesis, EE Dept., March 1966, AD 631-396

TR-26 Cheek, Thomas Burrell
Design of a Low-Cost Character Generator for Remote Computer Displays, S.M. Thesis, EE Dept., March 1966, AD 631-269

TR-27 Edwards, Daniel James
OCAS - On-Line Cryptanalytic Aid System, S.M. Thesis, EE Dept., May 1966, AD 633-678

TR-28 Smith, Arthur Anshel
Input/Output in Time-Shared, Segmented, Multiprocessor Systems, S.M. Thesis, EE Dept., June 1966, AD 637-215

TR-29 Ivie, Evan Leon
Search Procedures Based on Measures of Relatedness between Documents, Ph.D. Dissertation, EE Dept., June 1966, AD 636-275

TR-30 Saltzer, Jerome Howard TRaffic Control in a Multiplexed Computer System, Sc.D. Thesis, EE Dept., July 1966, AD 635-966

TR-31 Smith, Donald L.
Models and Data Structures for Digital Logic Simulation, S.M. Thesis, EE Dept., August 1966, AD 637-192

TR-32 Teitelman, Warren
PILOT: A Step Toward Man-Computer Symbiosis, Ph.D. Dissertation, Math. Dept., September 1966, AD 638-446

TR-33 Norton, Lewis M, ADEPT - A Heuristic Program for Proving Theorems of Group Theory, Ph.D. Dissertation, Math. Dept., October 1966, AD 645-660

TR-34 Van Horn, Earl C., Jr.
Computer Design for Asynchronously Reproducible Multiprocessing, Ph.D. Dissertation, EE Dept., November 1966, AD 650-407

TR-35 Fenichel, Robert R.
An On-Line System for Algebraic Manipulation, Ph.D. Dissertation, Appl. Math. (Harvard), December 1966, AD 657-282

TR-36 Martin, William A.
Symbolic Mathematical Laboratory, Ph.D. Dissertation, EE Dept., January 1967, AD 657-283

TR-37 Guzman-Arenas, Adolfo
Some Aspects of Pattern Recognition by Computer, S.M. Thesis, EE Dept., February 1967, AD 656-041

TR-38 Rosenberg, Ronald C., Daniel W. Kennedy and Roger A. Humphrey
A Low-Cost Output Terminal For Time-Shared Computers, March 1967, AD 662-027

TR-39 Forte, Allen
Syntax-Based Analytic Reading of Musical Scores, April 1967, AD 661-806

TR-40 Miller, James R.
On-Line Analysis for Social Scientists, May 1967, AD 668-009

TR-41 Coons, Steven A.
Surfaces for Computer-Aided Design of Space Forms, June 1967, AD 663-504

TR-42 Liu, Chung L., Gabriel D. Chang and Richard E. Marks
Design and Implementation of a Table-Driven Compiler System, July 1967, AD 668-960

TR-43 Wilde, Daniel U.
Program Analysis by Digital Computer, Ph.D. Dissertation, EE Dept., August 1967, AD 662-224

TR-44 Gorry, G. Anthony
A System for Computer-Aided Diagnosis, Ph.D. Dissertation, Sloan School, September 1967, AD 662-665

TR-45 Leal-Cantu, Nestor
On the Simulation of Dynamic Systems with Lumped Parameters and Time Delays, S.M. Thesis, ME Dept., October 1967, AD 663-502

TR-46 Alsop, Joseph W.
A Canonic Translator, S.B. Thesis, EE Dept., November 1967, AD 663-503

TR-47 Moses, Joel
Symbolic Integration, Ph.D. Dissertation, Math. Dept., December 1967, AD 662-666

TR-48 Jones, Malcolm M.
Incremental Simulation on a Time-Shared Computer, Ph.D. Dissertation, Sloan School, January 1968, AD 662-225

TR-49 Luconi, Fred L.
Asynchronous Computational Structures, Ph.D. Dissertation, EE Dept., February 1968, AD 667-602

TR-50 Denning, Peter J.
Resource Allocation in Multiprocess Computer Systems, Ph.D. Dissertation, EE Dept., May 1968, AD 675-554

TR-51 Charniak, Eugene
CARPS, A Program which Solves Calculus Word Problems, S.M. Thesis, EE Dept., July 1968, AD 673-670

TR-52 Deitel, Harvey M.
Absentee Computations in a Multiple-Access Computer System, S.M. Thesis, EE Dept., August 1968, AD 684-738

TR-53 Slutz, Donald R.
The Flow Graph Schemata Model of Parallel Computation, Ph.D. Dissertation, EE Dept., September 1968, AD 683-393

TR-54 Grochow, Jerrold M.
The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System, S.M. Thesis, EE Dept., October 1968, AD 689-468

TR-55 Rappaport, Robert L.
Implementing Multi-Process Primitives in a Multiplexed Computer System, S.M. Thesis, EE Dept., November 1968, AD 689-469

TR-56 Thornhill, Daniel E., Robert H. Stotz, Douglas T. Ross and John E. Ward
An Integrated Hardware-Software System for Computer Graphics in Time-Sharing, December 1968, AD 685-202

TR-57 Morris, James H.
Lambda-Calculus Models of Programming Languages, Ph.D. Dissertation, Sloan School, December 1968, AD 683-394

TR-58 Greenbaum, Howard J.
A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer System, S.M. Thesis, EE Dept., January 1969, AD 686-988

TR-59 Guzman, Adolfo
Computer Recognition of Three- Dimensional Objects in a Visual Scene, Ph.D. Dissertation, EE Dept., December 1968, AD 692-200

TR-60 Ledgard, Henry F.
A Formal System for Defining the Syntax and Semantics of Computer Languages, Ph.D. Dissertation, EE Dept., April 1969, AD 689-305

TR-61 Baecker, Ronald M.
Interactive Computer-Mediated Animation, Ph.D. Dissertation, EE Dept., June 1969, AD 690-887

TR-62 Tillman, Coyt C., Jr.
EPS: An Interactive System for Solving Elliptic Boundary-Value Problems with Facilities for Data Manipulation and General-Purpose Computation, June 1969, AD 692-462

TR-63 Brackett, John W., Michael Hammer and Daniel E. Thornhill
Case Study in Interactive Graphics Programming: A Circuit Drawing and Editing Program for Use with a Storage-Tube Display Terminal, October 1969, AD 699-930

TR-64 Rodriguez, Jorge E.
A Graph Model for Parallel Computations, Sc.D. Thesis, EE Dept., September 1969, AD 697-759

TR-65 DeRemer, Franklin L.
Practical Translators for LR(k) Languages, Ph.D. Dissertation, EE Dept., October 1969, AD 699-501

TR-66 Beyer, Wendell T.
Recognition of Topological Invariants by Iterative Arrays, Ph.D. Dissertation, Math. Dept., October 1969, AD 699-502

TR-67 Vanderbilt, Dean H.
Controlled Information Sharing in a Computer Utility, Ph.D. Dissertation, EE Dept., October 1969, AD 699-503

TR-68 Selwyn, Lee L.
Economies of Scale in Computer Use: Initial Tests and Implications for The Computer Utility, Ph.D. Dissertation, Sloan School, June 1970, AD 710-011

TR-69 Gertz, Jeffrey L.
Hierarchical Associative Memories for Parallel Computation, Ph.D. Dissertation, EE Dept., June 1970, AD 711-091

TR-70 Fillat, Andrew I. and Leslie A. Kraning
Generalized Organization of Large Data-Bases: A Set-Theoretic Approach to Relations, S.B. & S.M. Thesis, EE Dept., June 1970, AD 711-060

TR-71 Fiasconaro, James G.
A Computer-Controlled Graphical Display Processor, S.M. Thesis, EE Dept., June 1970, AD 710-479

TR-72 Patil, Suhas S.
Coordination of Asynchronous Events, Sc.D. Thesis, EE Dept., June 1970, AD 711-763

TR-73 Griffith, Arnold K.
Computer Recognition of Prismatic Solids, Ph.D. Dissertation, Math. Dept., August 1970, AD 712-069

TR-74 Edelberg, Murray
Integral Convex Polyhedra and an Approach to Integralization, Ph.D. Dissertation, EE Dept., August 1970, AD 712-070

TR-75 Hebalkar, Prakash G.
Deadlock-Free Sharing of Resources in Asynchronous Systems, Sc.D. Thesis, EE Dept., September 1970, AD 713-139

TR-76 Winston, Patrick H.
Learning Structural Descriptions from Examples, Ph.D. Dissertation, EE Dept., September 1970, AD 713-988

TR-77 Haggerty, Joseph P.
Complexity Measures for Language Recognition by Canonic Systems, S.M. Thesis, EE Dept., October 1970, AD 715-134

TR-78 Madnick, Stuart E.
Design Strategies for File Systems, S.M. Thesis, EE Dept. & Sloan School, October 1970, AD 714-269

TR-79 Horn, Berthold K.
Shape from Shading: A Method for Obtaining the Shape of a Smooth Opaque Object from One View, Ph.D. Dissertation, EE Dept., November 1970, AD 717-336

TR-80 Clark, David D., Robert M. Graham, Jerome H. Saltzer and Michael D. Schroeder
The Classroom Information and Computing Service, January 1971, AD 717-857

TR-81 Banks, Edwin R.
Information Processing and Transmission in Cellular Automata, Ph.D. Dissertation, ME Dept., January 1971, AD 717-951

TR-82 Krakauer, Lawrence J.
Computer Analysis of Visual Properties of Curved Objects, Ph.D. Dissertation, EE Dept., May 1971, AD 723-647

TR-83 Lewin, Donald E.
In-Process Manufacturing Quality Control, Ph.D. Dissertation, Sloan School, January 1971, AD 720-098

TR-84 Winograd, Terry
Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, Ph.D. Dissertation, Math. Dept., February 1971, AD 721-399

TR-85 Miller, Perry L.
Automatic Creation of a Code Generator from a Machine Description, E.E. Thesis, EE Dept., May 1971, AD 724-730

TR-86 Schell, Roger R.
Dynamic Reconfiguration in a Modular Computer System, Ph.D. Dissertation, EE Dept., June 1971, AD 725-859

TR-87 Thomas, Robert H.
A Model for Process Representation and Synthesis, Ph.D. Dissertation, EE Dept., June 1971, AD 726-049

TR-88 Welch, Terry A.
Bounds on Information Retrieval Efficiency in Static File Structures, Ph.D. Dissertation, EE Dept., June 1971, AD 725-429

TR-89 Owens, Richard C., Jr.
Primary Access Control in Large-Scale Time-Shared Decision Systems, S.M. Thesis, Sloan School, July 1971, AD 728-036

TR-90 Lester, Bruce P.
Cost Analysis of Debugging Systems, S.B. & S.M. Thesis, EE Dept., September 1971, AD 730-521

TR-91 Smoliar, Stephen W.
A Parallel Processing Model of Musical Structures, Ph.D. Dissertation, Math. Dept., September 1971, AD 731-690

TR-92 Wang, Paul S.
Evaluation of Definite Integrals by Symbolic Manipulation, Ph.D. Dissertation, Math. Dept., October 1971, AD 732-005

TR-93 Greif, Irene Gloria
Induction in Proofs about Programs, S.M. Thesis, EE Dept., February 1972, AD 737-701

TR-94 Hack, Michel Henri Theodore
Analysis of Production Schemata by Petri Nets, S.M. Thesis, EE Dept., February 1972, AD 740-320

TR-95 Fateman, Richard J.
Essays in Algebraic Simplification (A revision of a Harvard Ph.D. Dissertation), April 1972, AD 740-132

TR-96 Manning, Frank
Autonomous, Synchronous Counters Constructed Only of J-K Flip-Flops, S.M. Thesis, EE Dept., May 1972, AD 744-030

TR-97 Vilfan, Bostjan
The Complexity of Finite Functions, Ph.D. Dissertation, EE Dept., March 1972, AD 739-678

TR-98 Stockmeyer, Larry Joseph
Bounds on Polynomial Evaluation Algorithms, S.M. Thesis, EE Dept., April 1972, AD 740-328

TR-99 Lynch, Nancy Ann
Relativization of the Theory of Computational Complexity, Ph.D. Dissertation, Math. Dept., June 1972, AD 744-032

TR-100 Mandl, Robert
Further Results on Hierarchies of Canonic Systems, S.M. Thesis. EE Dept., June 1972, AD 744-206

TR-101 Dennis, Jack B.
On the Design and Specification of a Common Base Language, June 1972, AD 744-207

TR-102 Hossley, Robert F.
Finite Tree Automata and ω-Automata, S.M. Thesis, EE Dept., September 1972, AD 749-367

TR-103 Sekino, Akira
Performance Evaluation of Multiprogrammed Time-Shared Computer Systems, Ph.D. Dissertation, EE Dept., September 1972, AD 749-949

TR-104 Schroeder, Michael D.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D. Dissertation, EE Dept., September 1972, AD 750-173

TR-105 Smith, Burton J.
An Analysis of Sorting Networks, Sc.D. Thesis, EE Dept., October 1972, AD 751-614

TR-106 Rackoff, Charles W.
The Emptiness and Complementation Problems for Automata on Infinite Trees, S.M. Thesis, EE Dept., January 1973, AD 756<-248

TR-107 Madnick, Stuart E.
Storage Hierarchy Systems, Ph.D. Dissertation, EE Dept., April 1973, AD 760-001

TR-108 Wand, Mitchell
Mathematical Foundations of Formal Language Theory, Ph.D. Dissertation, Math. Dept., December 1973.

TR-109 Johnson, David S.
Near-Optimal Bin Packing Algorithms, Ph.D. Dissertation, Math. Dept., June 1973, PB 222-090

TR-110 Moll, Robert
Complexity Classes of Recursive Functions, Ph.D. Dissertation, Math. Dept., June 1973, AD 767-730

TR-111 Linderman, John P.
Productivity in Parallel Computation Schemata, Ph.D. Dissertation, EE Dept., December 1973, PB 226-159/AS

TR-112 Hawryszkiewycz, Igor T.
Semantics of Data Base Systems, Ph.D. Dissertation, EE Dept., December 1973, PB 226-061/AS

TR-113 Herrmann, Paul P.
On Reducibility Among Combinatorial Problems, S.M. Thesis, Math. Dept., December 1973, PB 226-157/AS

TR-114 Metcalfe, Robert M.
Packet Communication, Ph.D. Dissertation, Applied Math., Harvard University, December 1973, AD 771-430

TR-115 Rotenberg, Leo
Making Computers Keep Secrets, Ph.D. Dissertation, EE Dept., February 1974, PB 229-352/AS

TR-116 Stern, Jerry A.
Backup and Recovery of On-Line Information in a Computer Utility, S.M. & E.E. Thesis, EE Dept., January 1974, AD 774-141

TR-117 Clark, David D.
An Input/Output Architecture for Virtual Memory Computer Systems, Ph.D. Dissertation, EE Dept., January 1974, AD 774-738

TR-118 Briabrin, Victor
An Abstract Model of a Research Institute: Simple Automatic Programming Approach, March 1974, PB 231-505/AS

TR-119 Hammer, Michael M.
A New Grammatical Transformation into Deterministic Top-Down Form, Ph.D. Dissertation, EE Dept., February 1974, AD 775-545

TR-120 Ramchandani, Chander
Analysis of Asynchronous Concurrent Systems by Timed Petri Nets, Ph.D. Dissertation, EE Dept., February 1974, AD 775-618

TR-121 Yao, Foong F.
On Lower Bounds for Selection Problems, Ph.D. Dissertation, Math. Dept., March 1974, PB 230-950/AS

TR-122 Scherf, John A.
Computer and Data Security: A Comprehensive Annotated Bibliography, S.M. Thesis, Sloan School, January 1974, AD 775-546

TR-123 Introduction to Multics
February 1974, AD 918-562

TR-124 Laventhal, Mark S.
Verification of Programs Operating on Structured Data, S.B. & S.M. Thesis, EE Dept., March 1974, PB 231-365/AS

TR-125 Mark, William S.
A Model-Debugging System, S.B. & S.M. Thesis, EE Dept., April 1974, AD 778-688

TR-126 Altman, Vernon E.
A Language Implementation System, S.B. & S.M. Thesis, Sloan School, May 1974, AD 780-672

TR-127 Greenberg, Bernard S.
An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory, S.M. Thesis, EE Dept., May 1974, AD 780-407

TR-128 Frankston, Robert M.
The Computer Utility as a Marketplace for Computer Services, S.M. & E.E. Thesis, EE Dept., May 1974, AD 780-436

TR-129 Weissberg, Richard W.
Using Interactive Graphics in Simulating the Hospital Emergency Room, S.M. Thesis, EE Dept., May 1974, AD 780-437

TR-130 Ruth, Gregory R.
Analysis of Algorithm Implementations, Ph.D. Dissertation, EE Dept., May 1974, AD 780-408

TR-131 Levin, Michael
Mathematical Logic for Computer Scientists, June 1974.

TR-132 Janson, Philippe A.
Removing the Dynamic Linker from the Security Kernel of a Computing Utility, S.M. Thesis, EE Dept., June 1974, AD 781-305

TR-133 Stockmeyer, Larry J.
The Complexity of Decision Problems in Automata Theory and Logic, Ph.D. Dissertation, EE Dept., July 1974, PB 235-283/AS

TR-134 Ellis, David J.
Semantics of Data Structures and References, S.M. & E.E. Thesis, EE Dept., August 1974, PB 236-594/AS

TR-135 Pfister, Gregory F.
The Computer Control of Changing Pictures, Ph.D. Dissertation, EE Dept., September 1974, AD 787-795

TR-136 Ward, Stephen A.
Functional Domains of Applicative Languages, Ph.D. Dissertation, EE Dept., September 1974, AD 787-796

TR-137 Seiferas, Joel I.
Nondeterministic Time and Space Complexity Classes, Ph.D. Dissertation, Math. Dept., September 1974.
PB 236-777/AS

TR-138 Yun, David Y. Y.
The Hensel Lemma in Algebraic Manipulation, Ph.D. Dissertation, Math. Dept., November 1974, AD A002-737

TR-139 Ferrante, Jeanne
Some Upper and Lower Bounds on Decision Procedures in Logic, Ph.D. Dissertation, Math. Dept., November 1974.
PB 238-121/AS

TR-140 Redell, David D.
Naming and Protection in Extendible Operating Systems, Ph.D. Dissertation, EE Dept., November 1974, AD A001-721

TR-141 Richards, Martin, A. Evans and R. Mabee
The BCPL Reference Manual, December 1974, AD A003-599

TR-142 Brown, Gretchen P.
Some Problems in German to English Machine Translation, S.M. & E.E. Thesis, EE Dept., December 1974, AD A003-002

TR-143 Silverman, Howard
A Digitalis Therapy Advisor, S.M. Thesis, EE Dept., January 1975.

TR-144 Rackoff, Charles
The Computational Complexity of Some Logical Theories, Ph.D. Dissertation, EE Dept., February 1975.

TR-145 Henderson, D. Austin
The Binding Model: A Semantic Base for Modular Programming Systems, Ph.D. Dissertation, EE Dept., February 1975, AD A006-961

TR-146 Malhotra, Ashok
Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis, Ph.D. Dissertation, EE Dept., February 1975.

TR-147 Van De Vanter, Michael L.
A Formalization and Correctness Proof of the CGOL Language System, S.M. Thesis, EE Dept., March 1975.

TR-148 Johnson, Jerry
Program Restructuring for Virtual Memory Systems, Ph.D. Dissertation, EE Dept., March 1975, AD A009-218

TR-149 Snyder, Alan
A Portable Compiler for the Language C, S.B. & S.M. Thesis, EE Dept., May 1975, AD A010-218

TR-150 Rumbaugh, James E.
A Parallel Asynchronous Computer Architecture for Data Flow Programs, Ph.D. Dissertation, EE Dept., May 1975, AD A010-918

TR-151 Manning, Frank B.
Automatic Test, Configuration, and Repair of Cellular Arrays, Ph.D. Dissertation, EE Dept., June 1975, AD A012-822

TR-152 Qualitz, Joseph E.
Equivalence Problems for Monadic Schemas, Ph.D. Dissertation, EE Dept., June 1975, AD A012-823

TR-153 Miller, Peter B.
Strategy Selection in Medical Diagnosis, S.M. Thesis, EE & CS Dept., September 1975.

TR-154 Greif, Irene
Semantics of Communicating Parallel Processes, Ph.D. Dissertation, EE & CS Dept., September 1975, AD A016-302

TR-155 Kahn, Kenneth M.
Mechanization of Temporal Knowledge, S.M. Thesis, EE & CS Dept., September 1975.

TR-156 Bratt, Richard G.
Minimizing the Naming Facilities Requiring Protection in a Computer Utility, S.M. Thesis, EE & CS Dept., September 1975.

TR-157 Meldman, Jeffrey A.
A Preliminary Study in Computer-Aided Legal Analysis, Ph.D. Dissertation, EE & CS Dept., November 1975, AD A018-997

TR-158 Grossman, Richard W.
Some Data-base Applications of Constraint Expressions, S.M. Thesis, EE & CS Dept., February 1976, AD A024-149

TR-159 Hack, Michel
Petri Net Languages, March 1976.

TR-160 Bosyj, Michael
A Program for the Design of Procurement Systems, S.M. Thesis, EE & CS Dept., May 1976, AD A026-688

TR-161 Hack, Michel
Decidability Questions, Ph.D. Dissertation, EE & CS Dept., June 1976.

TR-162 Kent, Stephen T.
Encryption-Based Protection Protocols for Interactive User-Computer Communication, S.M. Thesis, EE & CS Dept., June 1976, AD A026-911

TR-163 Montgomery, Warren A.
A Secure and Flexible Model of Process Initiation for a Computer Utility, S.M. & E.E. Thesis, EE & CS Dept., June 1976.

TR-164 Reed, David P.
Processor Multiplexing in a Layered Operating System, S.M. Thesis, EE & CS Dept., July 1976.

TR-165 McLeod, Dennis J.
High Level Expression of Semantic Integrity Specifications in a Relational Data Base System, S.M. Thesis, EE & CS Dept., September 1976, AD A034-184

TR-166 Chan, Arvola Y.
Index Selection in a Self-Adaptive Relational Data Base Management System, S.M. Thesis, EE & CS Dept., September 1976, AD A034-185

TR-167 Janson, Philippe A.
Using Type Extension to Organize Virtual Memory Mechanisms, Ph.D. Dissertation, EE & CS Dept., September 1976.

TR-168 Pratt, Vaughan R.
Semantical Considerations on Floyd-Hoare Logic, September 1976.

TR-169 Safran, Charles, James F. Desforges and Philip N. Tsichlis
Diagnostic Planning and Cancer Management, September 1976.

TR-170 Furtek, Frederick C.
The Logic of Systems, Ph.D. Dissertation, EE & CS Dept., December 1976.

TR-171 Huber, Andrew R.
A Multi-Process Design of a Paging System, S.M. & E.E. Thesis, EE & CS Dept., December 1976.

TR-172 Mark, William S.
The Reformulation Model of Expertise, Ph.D. Dissertation, EE & CS Dept., December 1976, AD A035-397

TR-173 Goodman, Nathan
Coordination of Parallel Processes in the Actor Model of Computation, S.M. Thesis, EE & CS Dept., December 1976.

TR-174 Hunt, Douglas H.
A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem, S.M. & E.E. Thesis, EE & CS Dept., December 1976.

TR-175 Goldberg, Harold J.
A Robust Environment for Program Development, S.M. Thesis, EE & CS Dept., February 1977.

TR-176 Swartout, William R.
A Digitalis Therapy Advisor with Explanations, S.M. Thesis, EE & CS Dept., February 1977.

TR-177 Mason, Andrew H.
A Layered Virtual Memory Manager, S.M. & E.E. Thesis, EE & CS Dept., May 1977.

TR-178 Bishop, Peter B.
Computer Systems with a Very Large Address Space and Garbage Collection, Ph.D. Dissertation, EE & CS Dept., May 1977, AD A040-601

TR-179 Karger, Paul A.
Non-Discretionary Access Control for Decentralized Computing Systems, S.M. Thesis, EE & CS Dept., May 1977, AD A040-804

TR-180 Luniewski, Allen W.
A Simple and Flexible System Initialization Mechanism, S.M. & E.E. Thesis, EE & CS Dept., May 1977.

TR-181 Mayr, Ernst W.
The Complexity of the Finite Containment Problem for Petri Nets, S.M. Thesis, EE & CS Dept., June 1977 .

TR-182 Brown, Gretchen P.
A Framework for Processing Dialogue, June 1977, AD A042-370

TR-183 Jaffe, Jeffrey M.
Semilinear Sets and Applications, S.M. Thesis, EE & CS Dept., July 1977.

TR-184 Levine, Paul H.
Facilitating Interprocess Communication in a Heterogeneous Network Environment, S.B. & S.M. Thesis, EE & CS Dept., July 1977, AD A043-901

TR-185 Goldman, Barry
Deadlock Detection in Computer Networks, S.B. & S.M. Thesis, EE & CS Dept., September 1977, AD A047-025

TR-186 Ackerman, William B.
A Structure Memory for Data Flow Computers, S.M. Thesis, EE & CS Dept., September 1977, AD A047-026

TR-187 Long, William J.
A Program Writer, Ph.D. Dissertation, EE & CS Dept., November 1977, AD A047-595

TR-188 Bryant, Randal E.
Simulation of Packet Communication Architecture Computer Systems, S.M. Thesis, EE & CS Dept., November 1977, AD A048-290

TR-189 Ellis, David J.
Formal Specifications for Packet Communication Systems, Ph.D. Dissertation, EE & CS Dept., November 1977, AD A048-980

TR-190 Moss, J. Eliot B.
Abstract Data Types in Stack Based Languages, S.M. Thesis, EE & CS Dept., February 1978, AD A052-332

TR-191 Yonezawa, Akinori
Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, Ph.D. Dissertation, EE & CS Dept., January 1978, AD A051-149

TR-192 Niamir, Bahram
Attribute Partitioning in a Self-Adaptive Relational Database System, S.M. Thesis, EE & CS Dept., January 1978, AD A053-292

TR-193 Schaffert, J. Craig
A Formal Definition of CLU, S.M. Thesis, EE & CS Dept., January 1978

TR-194 Hewitt, Carl and Henry Baker, Jr.
Actors and Continuous Functionals, February 1978, AD A052-266

TR-195 Bruss, Anna R.
On Time-Space Classes and Their Relation to the Theory of Real Addition, S.M. Thesis, EE & CS Dept., March 1978

TR-196 Schroeder, Michael D., David D. Clark, Jerome H. Saltzer and Douglas H. Wells
Final Report of the Multics Kernel Design Project, March 1978

TR-197 Baker, Henry Jr.
Actor Systems for Real-Time Computation, Ph.D. Dissertation, EE & CS Dept., March 1978, AD A053-328

TR-198 Halstead, Robert H., Jr.
Multiple-Processor Implementation of Message-Passing Systems, S.M. Thesis, EE & CS Dept., April 1978, AD A054-009

TR-199 Terman, Christopher J.
The Specification of Code Generation Algorithms, S.M. Thesis, EE & CS Dept., April 1978, AD A054-301

TR-200 Harel, David
Logics of Programs: Axiomatics and Descriptive Power, Ph.D. Dissertation, EE & CS Dept., May 1978

TR-201 Scheifler, Robert W.
A Denotational Semantics of CLU, S.M. Thesis, EE & CS Dept., June 1978

TR-202 Principato, Robert N., Jr.
A Formalization of the State Machine Specification Technique, S.M. & E.E. Thesis, EE & CS Dept., July 1978

TR-203 Laventhal, Mark S.
Synthesis of Synchronization Code for Data Abstractions, Ph.D. Dissertation, EE & CS Dept., July 1978, AD A058-232

TR-204 Teixeira, Thomas J.
Real-Time Control Structures for Block Diagram Schemata, S.M. Thesis, EE & CS Dept., August 1978, AD A061-122

TR-205 Reed, David P.
Naming and Synchronization in a Decentralized Computer System, Ph.D. Dissertation, EE & CS Dept., October 1978, AD A061-407

TR-206 Borkin, Sheldon A.
Equivalence Properties of Semantic Data Models for Database Systems, Ph.D. Dissertation, EE & CS Dept., January 1979, AD A066-386

TR-207 Montgomery, Warren A.
Robust Concurrency Control for a Distributed Information System, Ph.D. Dissertation, EE & CS Dept., January 1979, AD A066-996

TR-208 Krizan, Brock C.
A Minicomputer Network Simulation System, S.B. & S.M. Thesis, EE & CS Dept., February 1979

TR-209 Snyder, Alan
A Machine Architecture to Support an Object-Oriented Language, Ph.D. Dissertation, EE & CS Dept., March 1979, AD A068-111

TR-210 Papadimitriou, Christos H.
Serializability of Concurrent Database Updates, March 1979

TR-211 Bloom, Toby
Synchronization Mechanisms for Modular Programming Languages, S.M. Thesis, EE & CS Dept., April 1979, AD A069-819

TR-212 Rabin, Michael O.
Digitalized Signatures and Public-Key Functions as Intractable as Factorization, March 1979

TR-213 Rabin, Michael O.
Probabilistic Algorithms in Finite Fields, March 1979

TR-214 McLeod, Dennis
A Semantic Data Base Model and Its Associated Structured User Interface, Ph.D. Dissertation, EE & CS Dept., March 1979, AD A068-112

TR-215 Svobodova, Liba, Barbara Liskov and David Clark
Distributed Computer Systems: Structure and Semantics, April 1979, AD A070-286

TR-216 Myers, John M.
Analysis of the SIMPLE Code for Dataflow Computation, June 1979

TR-217 Brown, Donna J.
Storage and Access Costs for Implementations of Variable - Length Lists, Ph.D. Dissertation, EE & CS Dept., June 1979

TR-218 Ackerman, William B. and Jack B. Dennis
VAL--A Value-Oriented Algorithmic Language: Preliminary Reference Manual, June 1979, AD A072-394

TR-219 Sollins, Karen R.
Copying Complex Structures in a Distributed System, S.M. Thesis, EE & CS Dept., July 1979, AD A072-441

TR-220 Kosinski, Paul R.
Denotational Semantics of Determinate and Non-Determinate Data Flow Programs, Ph.D. Dissertation, EE & CS Dept., July 1979

TR-221 Berzins, Valdis A.
Abstract Model Specifications for Data Abstractions, Ph.D. Dissertation, EE & CS Dept., July 1979

TR-222 Halstead, Robert H., Jr.
Reference Tree Networks: Virtual Machine and Implementation, Ph.D. Dissertation, EE & CS Dept., September 1979, AD A076-570

TR-223 Brown, Gretchen P.
Toward a Computational Theory of Indirect Speech Acts, October 1979, AD A077-065

TR-224 Isaman, David L.
Data-Structuring Operations in Concurrent Computations, Ph.D. Dissertation, EE & CS Dept., October 1979

TR-225 Liskov, Barbara, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler and Alan Snyder
CLU Reference Manual, October 1979, AD A077-018

TR-226 Reuveni, Asher
The Event Based Language and Its Multiple Processor Implementations, Ph.D. Dissertation, EE & CS Dept., January 1980, AD A081-950

TR-227 Rosenberg, Ronni L.
Incomprehensible Computer Systems: Knowledge Without Wisdom, S.M. Thesis, EE & CS Dept., January 1980

TR-228 Weng, Kung-Song
An Abstract Implementation for a Generalized Data Flow Language, Ph.D. Dissertation, EE & CS Dept., January 1980

TR-229 Atkinson, Russell R.
Automatic Verification of Serializers, Ph.D. Dissertation, EE & CS Dept., March 1980, AD A082-885

TR-230 Baratz, Alan E.
The Complexity of the Maximum Network Flow Problem, S.M. Thesis, EE & CS Dept., March 1980

TR-231 Jaffe, Jeffrey M.
Parallel Computation: Synchronization, Scheduling, and Schemes, Ph.D. Dissertation, EE & CS Dept., March 1980

TR-232 Luniewski, Allen W.
The Architecture of an Object Based Personal Computer, Ph.D. Dissertation, EE & CS Dept., March 1980, AD A083-433

TR-233 Kaiser, Gail E.
Automatic Extension of an Augmented Transition Network Grammar for Morse Code Conversations, S.B. Thesis, EE & CS Dept., April 1980, AD A084-411

TR-234 Herlihy, Maurice P. TRansmitting Abstract Values in Messages, S.M. Thesis, EE & CS Dept., May 1980, AD A086-984

TR-235 Levin, Leonid A.
A Concept of Independence with Applications in Various Fields of Mathematics, May 1980

TR-236 Lloyd, Errol L.
Scheduling Task Systems with Resources, Ph.D. Dissertation, EE & CS Dept., May 1980

TR-237 Kapur, Deepak
Towards a Theory for Abstract Data Types, Ph.D. Dissertation, EE & CS Dept., June 1980, AD A085-877

TR-238 Bloniarz, Peter A.
The Complexity of Monotone Boolean Functions and an Algorithm for Finding Shortest Paths in a Graph, Ph.D. Dissertation, EE & CS Dept., June 1980

TR-239 Baker, Clark M.
Artwork Analysis Tools for VLSI Circuits, S.M. & E.E. Thesis, EE & CS Dept., June 1980, AD A087-040

PUBLICATIONS

TR-240 Montz, Lynn B.
Safety and Optimization Transformations for Data Flow Programs, S.M. Thesis, EE & CS Dept., July 1980

TR-241 Archer, Rowland F., Jr.
Representation and Analysis of Real-Time Control Structures, S.M. Thesis, EE & CS Dept., August 1980, AD A089-828

TR-242 Loui, Michael C.
Simulations Among Multidimensional Turing Machines, Ph.D. Dissertation, EE & CS Dept., August 1980

TR-243 Svobodova, Liba
Management of Object Histories in the Swallow Repository, August 1980, AD A089-836

TR-244 Ruth, Gregory R.
Data Driven Loops, August 1980

TR-245 Church, Kenneth W.
On Memory Limitations in Natural Language Processing, S.M. Thesis, EE & CS Dept., September 1980

TR-246 Tiuryn, Jerzy
A Survey of the Logic of Effective Definitions, October 1980

TR-247 Weihl, William E.
Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables, S.B.& S.M.Thesis, EE & CS Dept., October 1980

TR-248 LaPaugh, Andrea S.
Algorithms for Integrated Circuit Layout: An Analytic Approach, Ph.D.Dissertation, EE & CS Dept., November 1980

TR-249 Turkle, Sherry
Computers and People: Personal Computation, December 1980

TR-250 Leung, Clement Kin Cho
Fault Tolerance in Packet Communication Computer Architectures, Ph.D. Dissertation, EE & CS Dept., December 1980

TR-251 Swartout, William R.
Producing Explanations and Justifications of Expert Consulting Programs, Ph.D. Dissertation, EE & CS Dept., January 1981

TR-252 Arens, Gail C.
Recovery of the Swallow Repository, S.M. Thesis, EE & CS Dept., January 1981, AD A096-374

TR-253 Ilson, Richard
An Integrated Approach to Formatted Document Production, S.M. Thesis, EE & CS Dept., February 1981

TR-254 Ruth, Gregory, Steve Alter and William Martin
A Very High Level Language for Business Data Processing, March 1981

TR-255 Kent, Stephen T.
Protecting Externally Supplied Software in Small Computers, Ph.D. Dissertation, EE & CS Dept., March 1981

TR-256 Faust, Gregory G.
Semiautomatic Translation of COBOL into HIBOL, S.M. Thesis, EE & CS Dept., April 1981

Progress Reports

Project MAC Progress Report I, to July 1964
AD 465-088

Project MAC Progress Report II, July 1964-July 1965
AD 629-494

Project MAC Progress Report III, July 1965-July 1966
AD 648-346

Project Mac Progress Report IV, July 1966-July 1967
AD 681-342

Project MAC Progress Report V, July 1967-July 1968
AD 687-770

Project MAC Progress Report VI, July 1968-July 1969
AD 705-434

Project MAC Progress Report VII, July 1969-July 1970
AD 732-767

Project MAC Progress Report VIII, July 1970-July 1971
AD 735-148

Project MAC Progress Report IX, July 1971-July 1972
AD 756-689

Project MAC Progress Report X, July 1972-July 1973
AD 771-428

Project MAC Progress Report XI, July 1973-July 1974
AD A004-966

Laboratory for Computer Science Progress Report XII,
July 1974-July 1975, AD A024-527

Laboratory for Computer Science Progress Report XIII,
July 1975-July 1976, AD A061-246

Laboratory for Computer Science Progress Report XIV,
July 1976-July 1977, AD A061-932

Laboratory for Computer Science Progress Report 15,
July 1977-July 1978, AD A073-958

Laboratory for Computer Science Progress Report 16,
July 1978-July 1979, AD A088-355

Laboratory for Computer Science Progress Report 17,
July 1979-July 1980, AD A093-384

Copies of all reports with AD and PB numbers listed in Publications may be secured from the National Technical Information Service, U.S. Department of Commerce, Reports Division, 5285 Port Royal Road, Springfield, Virginia 22161. Prices vary. The AD or PB number must be supplied with the request.

END

FILMED

6-85

DTIC